



Gradle IN ACTION

Gradle 实战

Benjamin Muschko

FOREWORD BY Hans Dockter

MANNING

目錄

关于本书	0
项目自动化	1
构建工具	1.1
Java 构建工具	1.2
Gradle下一代构建工具	2
为什么选择Gradle	2.1
Gradle强大的特性	2.2
连续传递的特性	2.3
安装Gradle	2.4
Gradle起步	2.5
使用命令行	2.6
开始你的第一个Gradle项目	3
介绍这个Gradle项目	3.1
构建Java项目	3.2
Gradle部署Web项目	3.3
Gradle包装器	3.4
构建脚本基础	4
构建块	4.1
管理任务	4.2
构建生命周期	4.3
依赖管理	5
快速预览	5.1
实战依赖管理	5.2
声明依赖	5.3
使用和配置仓库	5.4
多项目构建	6
项目模块化	6.1
多项目打包	6.2
配置子项目	6.3
拆分项目文件	6.4

自定义项目	6.5
总结	6.6
使用Gradle进行测试	7
自动化测试	7.1
测试Java应用	7.2
单元测试	7.3

又多了一份Gradle教程 : Gradle In Action

Gradle In Action(Gradle实战)中文版

- Gradle In Action中文版(Gradle中文教程), 目前正在翻译当中 欢迎大家一起加入
- 如果发现不通顺或者有歧义的地方, 可以在评论里指出来, 我会及时改正的
- [Github托管地址](#)
- [阅读地址](#)
- 我会开放权限给每一个加入的伙伴 (翻译或者校对), 请提前邮箱联系
ouyanglip@gmail.com

如何参与

任何问题都欢迎直接联系我 ouyanglip@gmail.com

第一步先找到自己想翻译的部分

第二步把代码**fork**到自己的仓库里

1. fork主仓库到自己的仓库里
2. 然后 git clone 到自己的电脑上
3. 推荐大家使用 Gitbook 专属的编辑器 [地址](#)
4. 如果是想学习Markdown 纯手工的同学可以参考这里学习 [地址1](#) [地址2](#)
5. 修改完 git commit 到自己的仓库
6. 给主仓库的 master 分支提交 pull request
7. 然后等待其他人的审核
8. 审核修改完成之后你翻译的部分就会被合并到书里

贡献者列表

成员	联系方式	Github
Lippi	ouyanglip@gmail.com	Github
Kary	kary@163.com	Github

项目自动化简介

想象一下没有自动化构建工具的场景

大部分的软件开发者都会面临下面的情形：

- 让IDE完成所有的工作. 用IDE来编码，导航到源代码、实现新特性、编译代码、重构代码、运行单元测试，一旦代码写完了，就按下编译按钮。一旦IDE提示没有编译错误测试通过，然后就把代码放入版本控制系统中以便与其他人分享。IDE是非常强大的工具，但是每个人都要安装一套标准的版本来执行上面介绍的任务，当你需要使用一个只有新版IDE才有的特性时，你就不得不更新到新版的IDE。
- 我的电脑上运行正常. 由于时间比较紧，Joe检查版本控制的代码发现编译不了，似乎是源代码中缺少了某个类，因此他联系了Tom，Tom非常困惑怎么代码在Joe的电脑上没办法编译成功，和Joe讨论完之后，他意识到自己忘记提交一个类到版本控制当中，所以无法编译成功，接下来整个团队都阻塞在这一步，直到Tom提交缺失的那个类上去。
- 代码集成简直就是个灾难. Acem有两个开发小组，一个集中于开发基于web的用户接口，另一小组集中开发服务器后台程序，当两个小组的人集中在一起测试整个程序时，发现程序的某些功能没有按照预期那样运行，一些链接无法解析或者直接返回错误的结果。
- 测试过程慢的像蜗牛. QA小组非常急切的接收第一版的app，可想而知，他们对低质量的程序是没什么耐心的，每次程序修改之后，都要进行相同的测试过程。小组停下来检查每次提交的改变，最新的版本是通过IDE构建的，代码传递到测试服务器，但是整个团队都在等待测试结果。

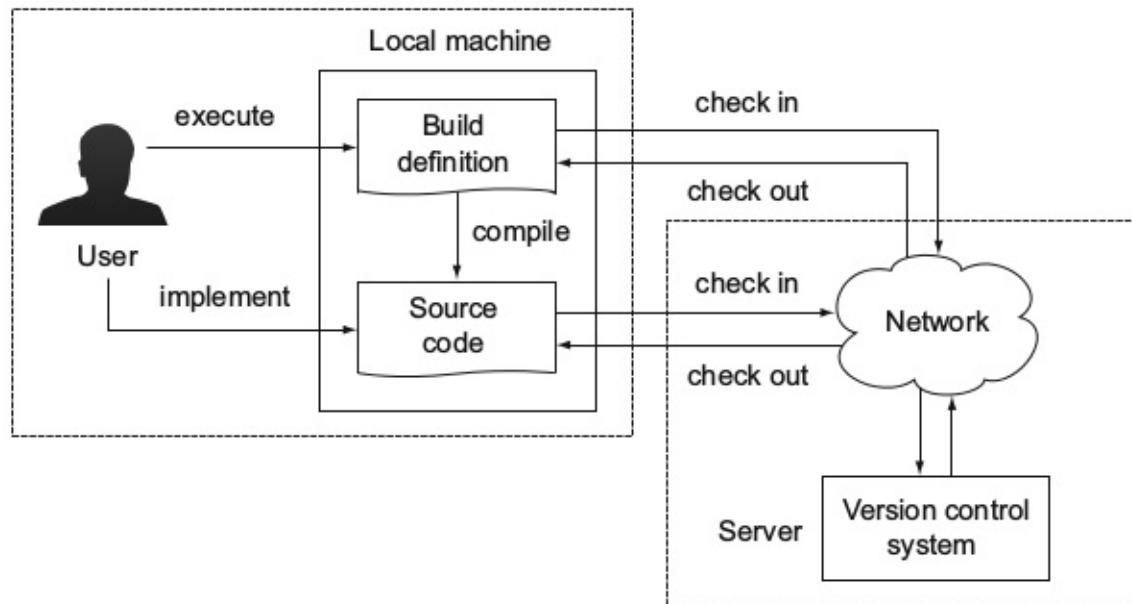
这时候你就需要一个自动化的构建工具。

项目自动化的优势

1. 避免手工介入
2. 创建可重复的构建过程
3. 使得构建非常便捷

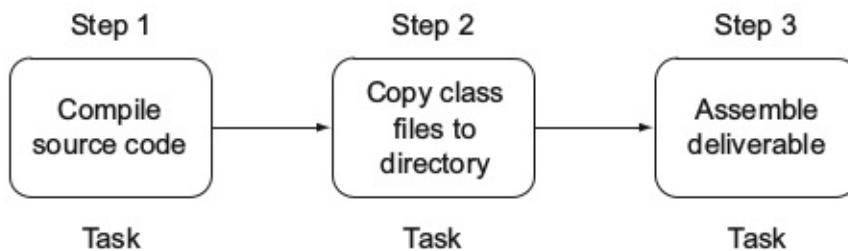
构建过程

大多数情况，用户在命令行执行一个脚本，脚本定义了任务执行的顺序，比如：编译源代码、从A路径复制文件到B路径、装配交付，这种自动化构建过程一天可能执行数次。



构建工具

你需要的就是一套工具，能把你自动化构建的需求表示成可执行的顺序的任务(tasks)，比如编译源代码，拷贝生成的class文件，组装交付。每一个任务都是一个工作单元，任务的顺序很重要，我们把任务和相互之间的依赖建模成一种有向无环图，比如下面这个：

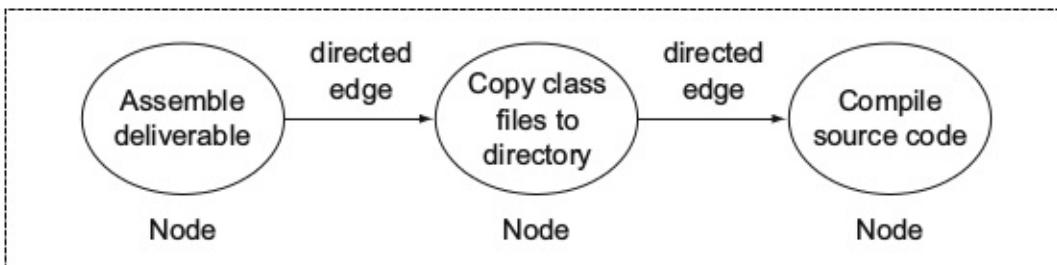


有向无环图

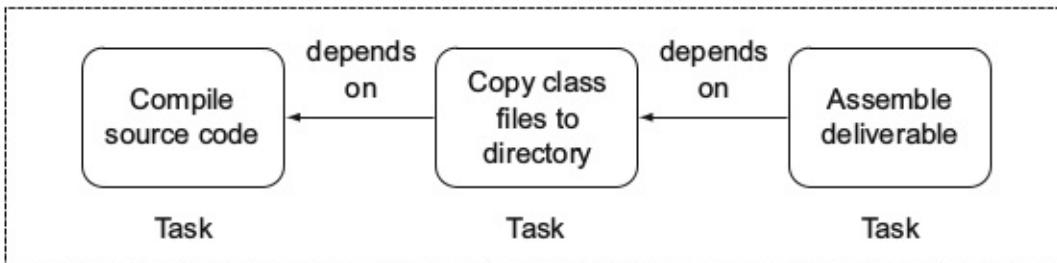
包含两个部分：

- 节点(node)：一个工作单元，在这里就是一个任务，比如编译源代码
- 边(edge): 一个有方向的边，表示相邻节点之间的依赖关系，如果一个任务定义了依赖，这个依赖的任务要在这个任务之前执行。

Directed acyclic graph



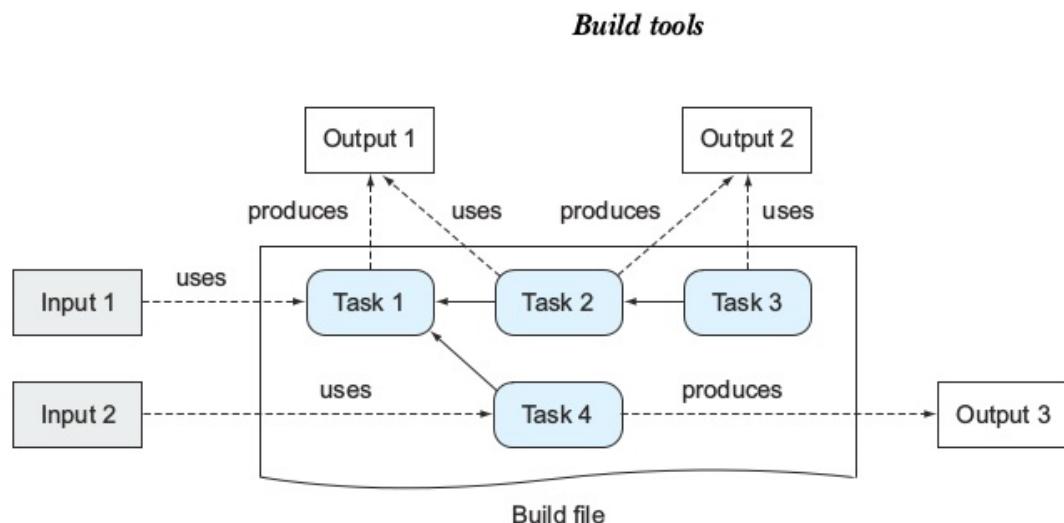
Task dependencies



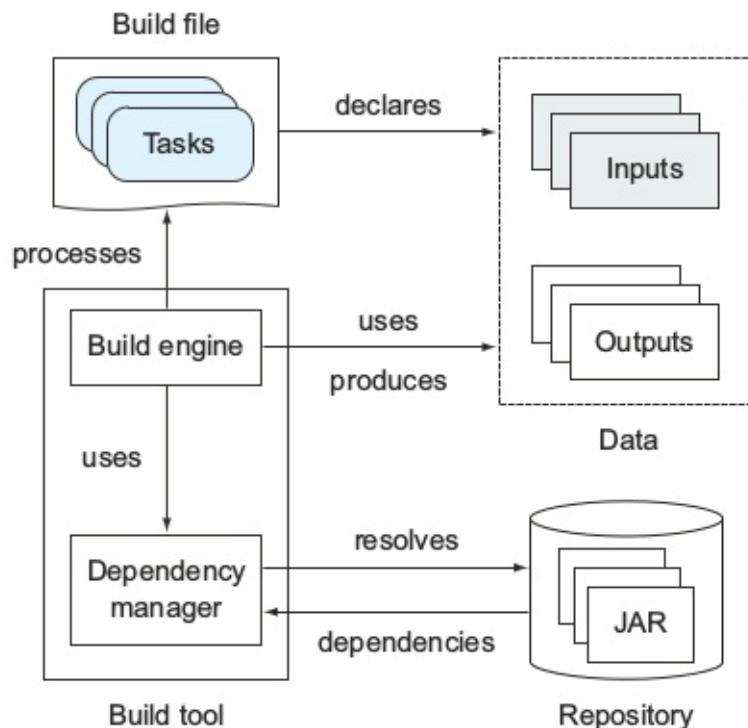
构建工具的组成

1. Build File. 包含构建需要的配置，定义了项目的依赖关系，比如第三方库的，以及以任务的形式存在的指令，定义了任务之间的先后顺序。

2. Build inputs and outputs: 任务把输入经过一系列步骤后产生输出。



3. 依赖管理。

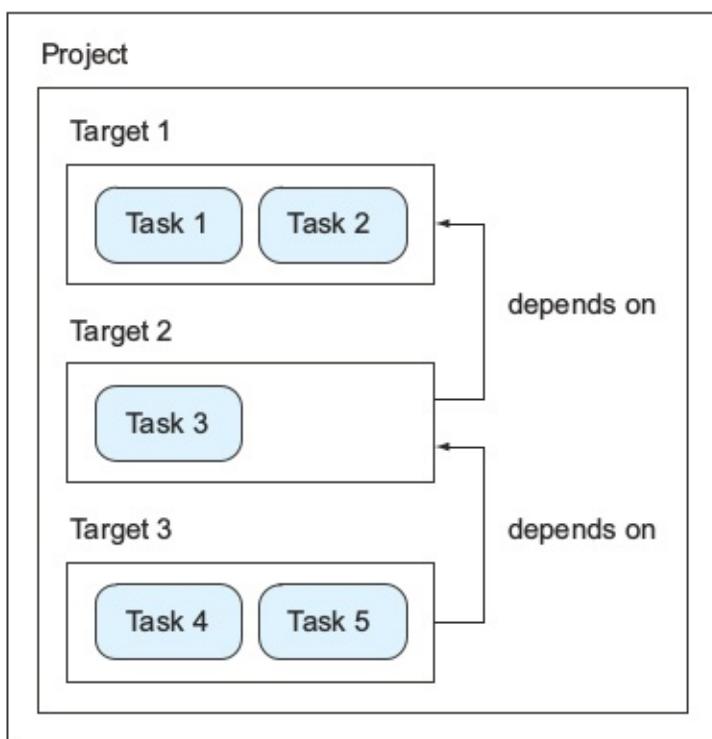


java构建工具

Ant

Ant 是 Apache 组织下的一个跨平台的项目构建工具，它是一个基于任务和依赖的构建系统，是过程式的。开发者需要显示的指定每一个任务，每个任务包含一组由 XML 编码的指令，必须在指令中明确告诉 Ant 源码在哪里，结果字节码存储在哪里，如何将这些字节码打包成 JAR 文件。Ant 没有生命周期，你必须定义任务和任务之间的依赖，还需要手工定义任务的执行序列和逻辑关系。这就无形中造成了大量的代码重复。

Build script



maven

Maven 是 Apache 组织下的一个跨平台的项目管理工具，它主要用来帮助实现项目的构建、测试、打包和部署。Maven 提供了标准的软件生命周期模型和构建模型，通过配置就能对项目进行全面的管理。它的跨平台性保证了在不同的操作系统上可以使用相同的命令来完成相应的任务。Maven 将构建的过程抽象成一个个的生命周期过程，在不同的阶段使用不同的已实现插件来完成相应的实际工作，这种设计方法极大的避免了设计和脚本编码的重复，极大的实现了复用。

Maven 不仅是一个项目构建工具还是一个项目管理工具。它有约定的目录结构（表 1）和生命周期，项目构建的各阶段各任务都由插件实现，开发者只需遵照约定的目录结构创建项目，再配置文件中生命项目的基本元素，Maven 就会按照顺序完成整个构建过程。Maven 的

这些特性在一定程度上大大减少了代码的重复。

为什么选择**Gradle**

如果你曾经使用过构建工具，你可能会对遇到的问题感到很沮丧，构建工具不是应该自动帮你完成项目的构建吗？你不得不向性能、扩展性等妥协。

比如你在构建一个项目的发布版本时，你要把一个文件拷贝到指定的位置，你在项目的元数据那里添加了版本的描述，如果版本号匹配一个特定的数字时，就把文件从A拷贝到B处。如果你依赖XML来构建，你要实现这个任务就像噩梦一样，你只能通过非标准的机制来添加一些脚本到构建中，结果就是把XML和脚本混在一起，随着时间的推移，你会添加越来越多的自定义的代码，结果就是项目越来越复杂很难维护。为什么不考虑用表达式的语言来定义你的构建逻辑呢？

另外一个例子，Maven跟随约定优于配置的规范，引入了标准化的项目布局和构建生命周期，给很多项目确保一个统一的结构这是个不错的方法。然而你手上的项目刚好和传统的约定不一样。Maven的一个严格的规定就是每个项目都要生成一个artifact，比如jar文件，但是你怎么从同一个源代码结构中创建两个不同的JAR文件，因此你不得不分开创建两个项目。

Java构建工具的发展

最早出现的是Ant，Ant里的每一个任务（target）都可以互相依赖，Ant的最大缺点就是依赖的外部库也要添加到版本控制系统中，因为Ant没有一个机制来把这些jar文件放在一个中央库里面，结果就是不断的拷贝和粘贴代码。

随后Maven在2004年出现了，Maven引入了标准的项目和路径结构，还有依赖管理，不幸的是自定义的逻辑很难实现，唯一的方法就是引入插件。

随后Ant通过Apache Ivy引入依赖管理来跟上Maven的脚步，Ant和Ivy集成实现了声明式的依赖，比如项目的编译和打包过程

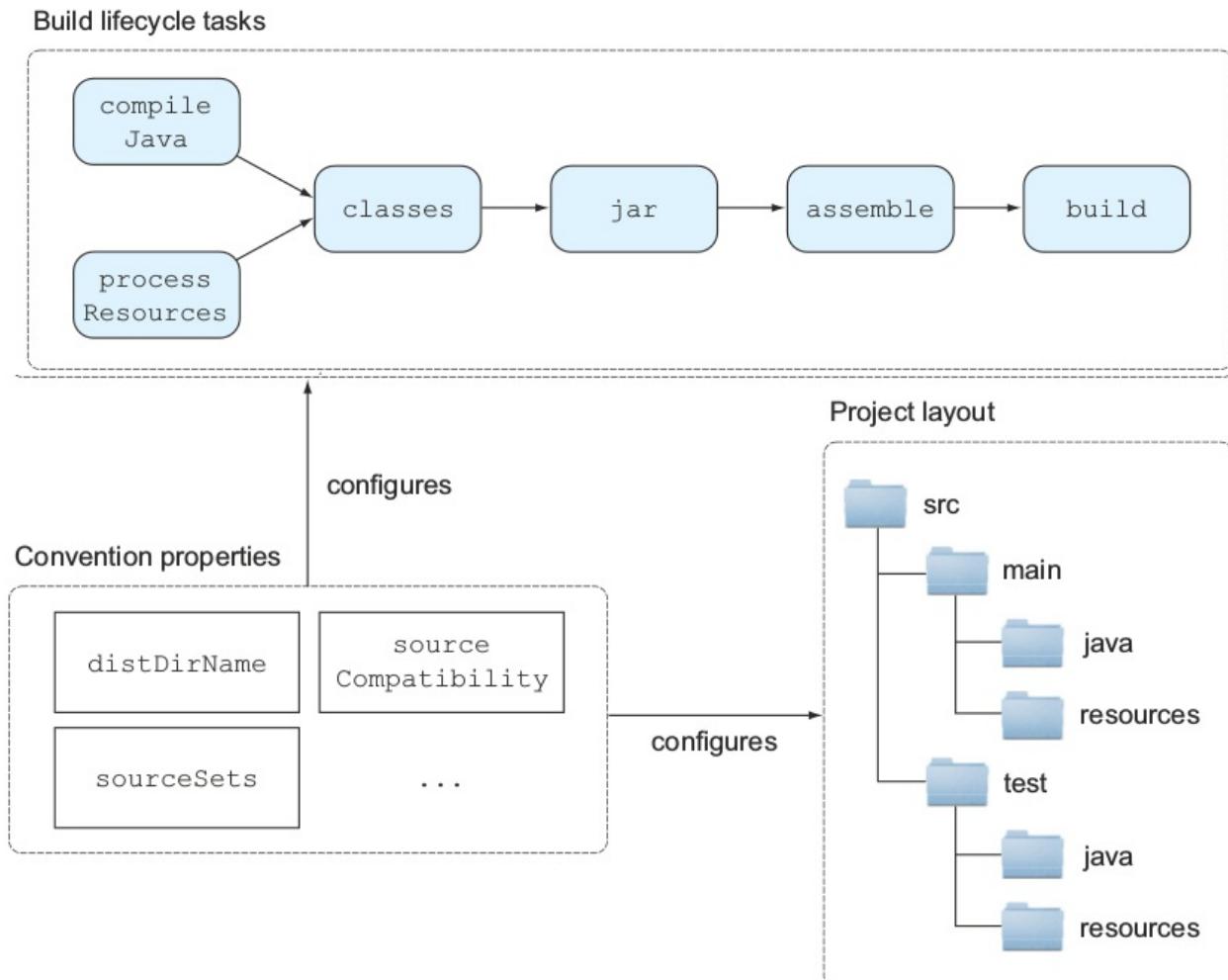
Gradle的出现满足了很多现在构建工具的需求，Gradle提供了一个DSL(领域特定语言)，一个约定优于配置的方法，还有更强大的依赖管理，Gradle使得我们可以抛弃XML的繁琐配置，引入动态语言Groovy来定义你的构建逻辑。

为什么要选择**Gradle**

假如你是一个开发者，项目自动构建是你每天工作的一部分，难道你就不想让你的构建代码和你写的源代码一样可以扩展、测试和维护？Gradle的构建脚本是声明式的、可读的，可以清晰的表达意图。使用Groovy代替XML来写代码大大减少了构建代码的大小。更重要的是，Gradle集成了其他构建工具，比如Ant和Maven，使得原来的项目很容易迁徙到Gradle。

Gradle提供了一些默认的Tasks给Java项目，比如，编译源代码、运行测试、打包JAR.每一个Java项目都有一个标准的路径布局，这个布局定义了去哪里找项目的源代码、资源文件和测试代码，你也可以在配置中修改这些默认位置。

Gradle的约定类似于Maven的约定优于配置的实现，Maven的约定就是一个项目只包含一个Java源代码路径，只产生一个JAR文件，对于企业级开发来讲这样是显然不够的，Gradle允许你打破传统的观念，Gradle的构建生命周期如下图：



和其他构建工具集成

Gradle完全兼容Ant、Maven，你可以很容易的从Ant或Maven迁移到Gradle，Gradle并不强迫你完全把你的Build逻辑迁移过来，它允许你复用已有的Ant构建逻辑。Gradle完全兼容Maven和Ivy仓库，你可以从中检索依赖也可以发布你的文件到仓库中，Gradle提供转换器能把Maven的构建逻辑转换成Gradle的构建脚本。

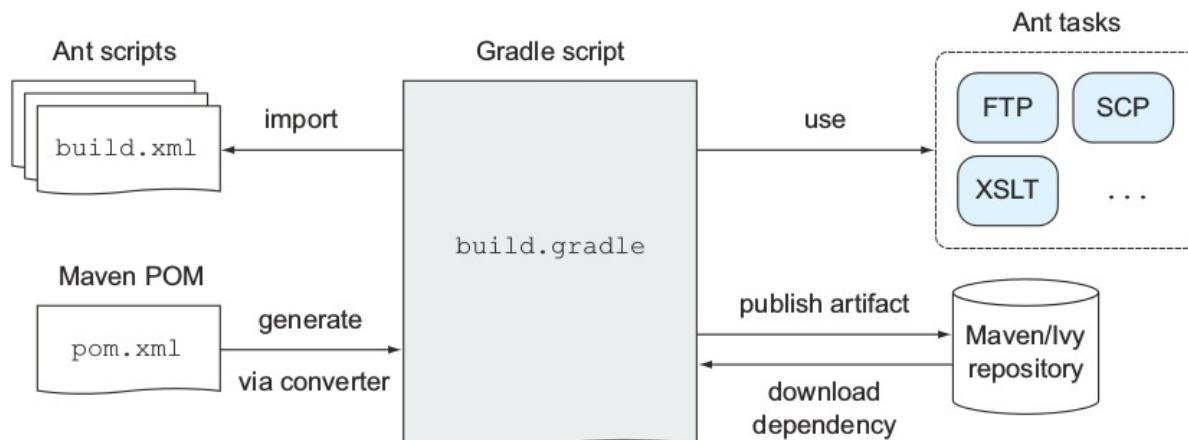
从Ant,Maven迁移到Gradle

现有的Ant脚本可以无缝的导入到Gradle项目中，Ant的Target在运行时直接映射成Gradle的任务，Gradle有一个AntBuilder可以把你的Ant脚本混成Gradle的DSL（领域特定语言），这些脚本看起来像是Ant的XML，但是去掉了尖括号，对于Ant用户来说非常方便，不需要担心过渡到Gradle的学习周期。

Gradle能够解析现有的Maven POM，从而得到传递性依赖的信息，并且引入到当前项目中，在此基础上，它也支持排除传递性依赖或者干脆关闭传递性依赖，这一点是Maven所不具备的特性。Gradle项目使用Maven项目生成的资源已经不是个问题了，接着需要反过来考虑，Maven用户是否能够使用Gradle生成的资源呢？或者更简单点问，Gradle项目生成的构件是否可以发布到Maven仓库中供人使用呢？这一点非常重要，因为如果做不到这一点，你可能就会丢失大量的用户。幸运的是Gradle再次给出了令人满意的答案。使用Gradle的Maven Plugin，用户就可以轻松地将项目构件上传到Maven仓库中：

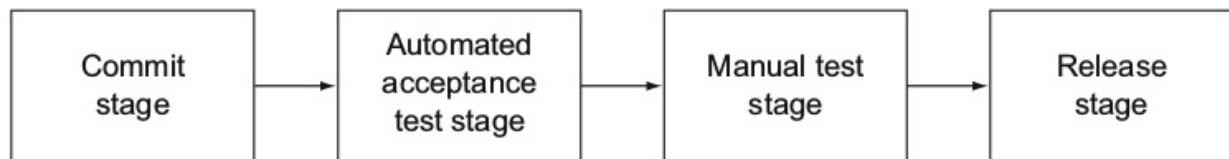
```
apply plugin: 'maven'
...
uploadArchives {
    repositories.mavenDeployer {
        repository(url: "http://localhost:8088/nexus/content/repositories/snapshots/") {
            authentication(userName: "admin", password: "admin123")
            pom.groupId = "com.juvenxu"
            pom.artifactId = "account-captcha"
        }
    }
}
```

在上传的过程中，Gradle能够基于build.gradle生成对应的Maven POM文件，用户可以自行配置POM信息，比如这里的groupId和artifactId，而诸如依赖配置这样的内容，Gradle是会自动帮你进行转换的。由于Maven项目之间依赖交互的直接途径就是仓库，而Gradle既能够使用Maven仓库，也能以Maven的格式将自己的内容发布到仓库中，因此从技术角度来说，即使在一个基于Maven的大环境中，局部使用Gradle也几乎不会是一个问题。



连续传递的特性

编译源代码只是整个过程的一个方面，更重要的是，你要把你的软件发布到生产环境中来产生商业价值，所以，你要运行测试，构建分布、分析代码质量、甚至为不同目标环境提供不同版本，然后部署。整个过程进行自动化操作是很有必要的。整个过程可以参考下图：



整个过程可以分成以下几个步骤：

- 编译源代码
- 运行单元测试和集成测试
- 执行静态代码分析、生成分析报告
- 创建发布版本
- 部署到目标环境
- 部署传递过程
- 执行冒烟测试和自动功能测试

安装Gradle

Gradle的安装非常方便，下载ZIP包，解压到本地目录，设置GRADLE_HOME环境变量并将GRADLE_HOME/bin加到PATH环境变量中，安装就完成了。用户可以运行gradle -v命令验证安装，这些初始的步骤和Maven没什么两样。我这里安装的Gradle版本是1.10，详细信息见下：

```
> gradle -v

-----
Gradle 1.10
-----
Build time: 2013-12-17 09:28:15 UTC
Build number: none
Revision: 36ced393628875ff15575fa03d16c1349ffe8bb6
Groovy: 1.8.6
Ant: Apache Ant(TM) version 1.9.2 compiled on July 8 2013
Ivy: 2.2.0
JVM: 1.7.0_45 (Oracle Corporation 24.45-b08)
OS: Mac OS X 10.9.2 x86_64
```

Gradle

Gradle目前的版本是2.4，根据其[Wiki上的Roadmap](#)，Gradle有着[让很多成熟项目都汗颜的文档](#)，其包括了安装指南、基本教程、以及一份近300页的全面用户指南。这对于用户来说是非常友好的，同时也说明了Gradle的开发者对这个项目非常有信心，要知道编写并维护文档可不是件轻松的工作，对于Gradle这样未来仍可能发生很大变动的项目来说尤为如此。

类似于Maven的 `pom.xml` 文件，每个Gradle项目都需要有一个对应的 `build.gradle` 文件，该文件定义一些任务（task）来完成构建工作，当然，每个任务是可配置的，任务之间也可以依赖，用户亦能配置缺省任务，就像这样：

```
defaultTasks 'taskB'

task taskA << {
    println "i'm task A"
}

task taskB << {
    println "i'm task B, and I depend on " + taskA.name
}

taskB.dependsOn taskA
```

运行命令\$ **gradle -q**之后（参数q让Gradle不要打印错误之外的日志），就能看到如下的预期输出：

```
i'm task A
i'm task B, and I depend on taskA
```

这不是和Ant如出一辙么？的确是这样，这种“任务”的概念与用法与Ant及其相似。Ant任务是Gradle世界的第一公民，Gradle对Ant做了很好的集成。除此之外，由于Gradle使用的Groovy脚本较XML更为灵活，因此，即使我自己不是Ant用户，我也仍然觉得Ant用户会喜欢上Gradle。

依赖管理和集成Maven仓库

我们知道依赖管理、仓库、约定优于配置等概念是Maven的核心内容，抛开其实现是否最优不谈，概念本身没什么问题，并且已经被广泛学习和接受。那Gradle实现了这些优秀概念了么？答案是肯定的。

先看依赖管理，我有一个简单的项目依赖于一些第三方类库包括SpringFramework、JUnit、Kaptcha等等。原来的Maven POM配置大概是这样的（篇幅关系，省略了部分父POM配置）：

```
<properties>
    <kaptcha.version>2.3</kaptcha.version>
</properties>

<dependencies>
    <dependency>
        <groupId>com.google.code.kaptcha</groupId>
        <artifactId>kaptcha</artifactId>
        <version>${kaptcha.version}</version>
        <classifier>jdk15</classifier>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-core</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-beans</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
    </dependency>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
    </dependency>
</dependencies>
```

然后我将其转换成Gradle脚本，结果是惊人的：

```

dependencies {
    compile('org.springframework:spring-core:2.5.6')
    compile('org.springframework:spring-beans:2.5.6')
    compile('org.springframework:spring-context:2.5.6')
    compile('com.google.code.kaptcha:kaptcha:2.3:jdk15')
    testCompile('junit:junit:4.7')
}

```

注意配置从原来的28行缩减至7行！这还不算我省略的一些父POM配置。依赖的groupId、artifactId、version、scope甚至是classifier，一点都不少。较之于Maven或者Ant的XML配置脚本，Gradle使用的Groovy脚本杀伤力太大了，爱美之心，人皆有之，相比于七旬老妇松松垮垮的皱纹，大家肯定都喜欢少女紧致的脸蛋，XML就是那老妇的皱纹。

关于Gradle的依赖管理起初我有一点担心，就是它是否有传递性依赖的机制呢？经过文档阅读和实际试验后，这个疑虑打消了，Gradle能够解析现有的Maven POM或者Ivy的XML配置，从而得到传递性依赖的信息，并且引入到当前项目中，这实在是一个聪明的做法。在此基础上，它也支持排除传递性依赖或者干脆关闭传递性依赖，其中第二点是Maven所不具备的特性。

自动化依赖管理的基石是仓库，Maven中央仓库已经成为了Java开发者不可或缺的资源，Gradle既然有依赖管理，那必然也得用到仓库，这当然也包括了Maven中央仓库，就像这样：

```

repositories {
    mavenLocal()
    mavenCentral()
    mavenRepo urls: "http://repository.sonatype.org/content/groups/
}

```

这段代码几乎不用解释，就是在Gradle中配置使用Maven本地仓库、中央仓库、以及自定义地址仓库。在我实际构建项目的时候，能看到终端打印的下载信息，下载后的文件被存储在 `USER_HOME/.gradle/cache/` 目录下供项目使用，这种实现的方法与Maven又是及其类似了，可以说Gradle不仅最大限度的继承Maven的很多理念，仓库资源也是直接拿来用。

Gradle项目使用Maven项目生成的资源已经不是个问题了，接着需要反过来考虑，Maven用户是否能够使用Gradle生成的资源呢？或者更简单点问，Gradle项目生成的构件是否可以发布到Maven仓库中供人使用呢？这一点非常重要，因为如果做不到这一点，你可能就会丢失大量的用户。幸运的是Gradle再次给出了令人满意的答案。使用Gradle的Maven Plugin，用户就可以轻松地将项目构件上传到Maven仓库中：

```
apply plugin: 'maven'  
...  
uploadArchives {  
    repositories.mavenDeployer {  
        repository(url: "http://localhost:8088/nexus/content/repositories/  
        authentication(userName: "admin", password: "admin123")  
        pom.groupId = "com.juvenxu"  
        pom.artifactId = "account-captcha"  
    }  
}  
}
```

在上传的过程中，Gradle能够基于 `build.gradle` 生成对应的Maven POM文件，用户可以自行配置POM信息，比如这里的`groupId`和`artifactId`，而诸如依赖配置这样的内容，Gradle是会自动帮你进行转换的。由于Maven项目之间依赖交互的直接途径就是仓库，而Gradle既能够使用Maven仓库，也能以Maven的格式将自己的内容发布到仓库中，因此从技术角度来说，即使在一个基于Maven的大环境中，局部使用Gradle也几乎不会是一个问题。

约定优于配置

如同Ant一般，Gradle给了用户足够的自由去定义自己的任务，不过同时Gradle也提供了类似Maven的约定优于配置方式，这是通过Gradle的Java Plugin实现的，从文档上看，Gradle是推荐这种方式的。Java Plugin定义了与Maven完全一致的项目布局：

- `src/main/java`
- `src/main/resources`
- `src/test/java`
- `src/test/resources`

区别在于，使用Groovy自定义项目布局更加的方便：

```

sourceSets {
    main {
        java {
            srcDir 'src/java'
        }
        resources {
            srcDir 'src/resources'
        }
    }
}

```

Gradle Java Plugin也定义了构建生命周期，包括编译主代码、处理资源、编译测试代码、执行测试、上传归档等等任务：

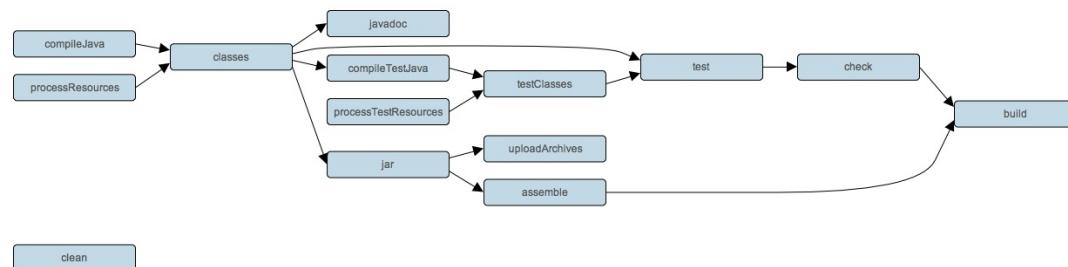


Figure 1. Gradle的构建生命周期

相对于Maven完全线性的生命周期，Gradle的构建生命周期略微复杂，不过也更为灵活，例如jar这个任务是用来打包的，它不像Maven那样依赖于执行测试的test任务，类似的，从图中可以看到，一个最终的build任务也没有依赖于uploadArchives任务。这个生命周期并没有将用户限制得很死，举个例子，我希望每次build都发布 SNAPSHOT版本到Maven仓库中，而且我只想使用最简单的\$ gradle clean build命令，那只需要添加一行任务依赖配置即可：

```
build.dependsOn 'uploadArchives'
```

由于Gradle完全是基于灵活的任务模型，因此很多事情包括覆盖现有任务，跳过任务都非常易于实现。而这些事情，在Maven的世界中，实现起来就比较麻烦，或者说Maven压根就不希望用户这么做。

小结

一番体验下来，Gradle给我最大的感觉是两点。其一是简洁，基于Groovy的紧凑脚本实在让人爱不释手，在表述意图方面也没有什么不清晰的地方。其二是灵活，各种在Maven中难以下手的事情，在Gradle就是小菜一碟，比如修改现有的构建生命周期，几行配置就完成了，

同样的事情，在Maven中你必须编写一个插件，那对于一个刚入门的用户来说，没个一两天几乎是不可能完成的任务。

不过即使如此，Gradle在未来能否取代Maven，在我看来也还是个未知数。它的一大障碍就是Groovy，几乎所有Java开发者都熟悉XML，可又有几个人了解Groovy呢？学习成本这道坎是很难跨越的，很多人抵制Maven就是因为学起来不容易，你现在让因为一个构建工具学习一门新语言（即使这门语言和Java非常接近），那得到冷淡的回复几乎是必然的事情。

Gradle的另外一个问题是它太灵活了，虽然它支持约定优于配置，不过从本文你也看到了，破坏约定是多么容易的事情。人都喜欢自由，爱自定义，觉得自己的需求是多么的特别，可事实上，从Maven的流行来看，几乎95%以上的情况你不需要自行扩展，如果你这么做了，只会让构建变得难以理解。从这个角度来看，自由是把双刃剑，Gradle给了你足够的自由，约定优于配置只是它的一个选项而已，这初看起来很诱人，却也可能使其重蹈Ant的覆辙。Maven在Ant的基础上引入了依赖管理、仓库以及约定优于配置等概念，是一个很大的进步，不过在我看来，Gradle并没有引入新的概念，给我感觉它是一个结合Ant和Maven理念的优秀实现。

如果你了解Groovy，也理解Maven的约定优于配置，那试试Gradle倒也不错，尤其是它几乎能和现有的Maven系统无缝集成，而且你也能享受到简洁带来的极大乐趣。其实说到简洁，也许在不久的将来Maven用户也能直接享受到，[Polyglot Maven](#)在这方面已经做了不少工作。

使用命令行操作

我们可以用Gradle命令来执行特定的任务，运行一个任务需要你知道该任务的名称，如果Gradle能够告诉你有哪些任务可以执行那岂不是很棒？Gradle提供了一个辅助的任务tasks来检查你的构建脚本，然后显示所有的任务，包含一个描述性的消息。

```
$ gradle -q tasks
```

输出如下：

```
All tasks runnable from root project

Build Setup tasks

setupBuild - Initializes a new Gradle build. [incubating]
wrapper - Generates Gradle wrapper files. [incubating]

Help tasks
-----
dependencies - Displays the dependencies of root project 'grouptherapy'.
dependencyInsight - Displays the insight into a specific dependency in root
- project 'grouptherapy'.
help - Displays a help message
projects - Displays the sub-projects of root project 'grouptherapy'.
properties - Displays the properties of root project 'grouptherapy'.
tasks - Displays the tasks runnable from root project 'grouptherapy' (some of
- the displayed tasks may belong to subprojects).

Other tasks
-----
groupTherapy

To see all tasks and more detail, run with --all.
```

Gradle提供任务组的概念，简而言之就是将一些任务归为一组，你可以执行这个组里面所有的任务，没有分组的任务在Other tasks，任务分组后面会讲到。

任务执行

要执行一个任务，只需要输入gradle + 任务名，Gradle确保这个任务和它所依赖的任务都会执行，要执行多个任务只需要在后面添加多个任务名。

任务名称缩写

Gradle提高效率的一个办法就是能够在命令行输入任务名的驼峰简写，当你的任务名称非常长的时候这很有用，当时你要确保你的简写必须是唯一确定那个任务，比如下面的情况：

```
task groupTherapy << {
...
}
task generateTests << {
...
}
```

这时候你使用gradle gT的时候Gradle就会报错，因为有多个任务匹配到gT \$ gradle yG0 gT
FAILURE: Could not determine which tasks to execute.

```
* What went wrong:
Task 'gT' is ambiguous in root project 'grouptherapy'. Candidates are:
- 'generateTests', 'groupTherapy'.
* Try:
Run gradle tasks to get a list of available tasks.
```

BUILD FAILED

运行的时候排除一个任务

比如运行的时候你要排除yayGradle0,你可以使用-x命令来完成

```
$ gradle groupTherapy -x yayGradle0
:yayGradle1
Gradle rocks
:yayGradle2
Gradle rocks
:groupTherapy
```

运行的时候Gradle排除了yayGradle0任务和它依赖的任务startSession。

命令行选项

- **-i:**Gradle默认不会输出很多信息，你可以使用-i选项改变日志级别为INFO
- **-s:**如果运行时错误发生打印堆栈信息
- **-q:**只打印错误信息

- `-?,-h,--help`: 打印所有的命令行选项
 - `-b,--build-file`: Gradle 默认执行 `build.gradle` 脚本, 如果想执行其他脚本可以使用这个命令, 比如 `gradle -b test.gradle`
 - `--offline`: 在离线模式运行 build, Gradle 只检查本地缓存中的依赖
 - `-D, --system-prop`: Gradle 作为 JVM 进程运行, 你可以提供一个系统属性比如: `-Dmyprop=myValue`
 - `-P,--project-prop`: 项目属性可以作为你构建脚本的一个变量, 你可以传递一个属性值给 build 脚本, 比如: `-Pmyprop=myValue`
-

- `tasks`: 显示项目中所有可运行的任务
- `properties`: 打印你项目中所有的属性值

第一个Gradle项目

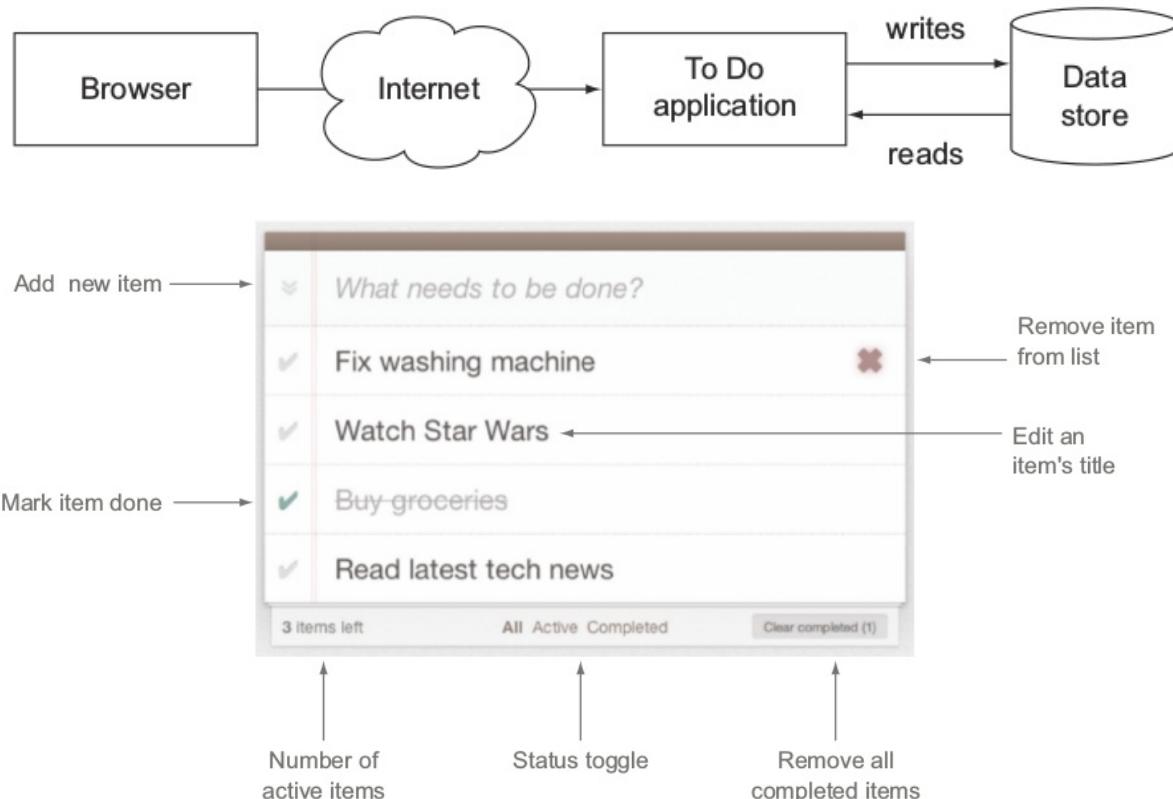
这一章将通过一个例子来介绍Gradle的强大特性，你将从中学到怎么用Gradle的标准插件来引导、配置和运行你的应用，这章结束的时候你应该对Gradle的工作机制有个清晰的认识。

The To Do Application

我们经常需要同时管理多个项目，有时候会发现多个项目很难维护达到了难以控制的地步，为了摆脱这个困惑，我们可以维护一个to-do列表，显然你可以把你所有要完成的任务写在一张纸上，当时如果能够随时随地查询你要完成的任务岂不更方便？

任务管理的情形

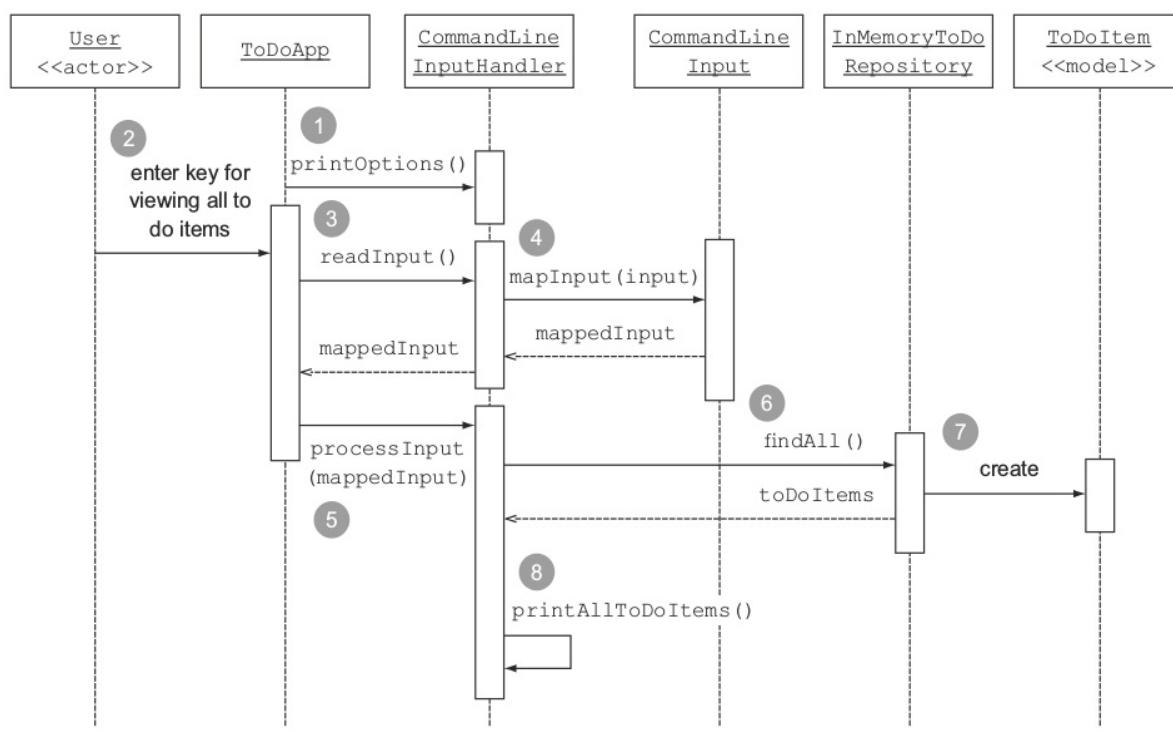
现在你知道了你的最终目的，每一个任务管理系统都是由一系列的任务组成的，任务通常有一个标题，任务可以添加到任务列表中，可以标记任务的完成状态以及删除任务，如下图所示：



实现用户交互功能

我们发现这个TO DO 应用包含典型的创建、读取、更新、删除操作(CRUD)，要持久化数据，你需要用一个模型来给任务建模，我们创建一个叫ToDoItem的Java类，一个POJO对象，为了简化这个应用，我们这里不采用数据库来存储，而是把数据存储在内存中，这很容易实现。实现存储接口的类是InMemoryToDoRespository,缺点就是你的应用程序关闭之后你就无法持久化数据了，后面我们会继续完善这个应用。

每一个标准的Java应用都有一个Main Class，应用程序的入口。这里的main class是ToDoApp,我们将会展现一栏的命令给用户选择，每一个命令被映射成一个枚举类型CommandLineInput,ComandLineInputHandler类用来处理用户输入执行相应的任务。下图显示了整个流程：



搭建应用的每一个模块

表示 Todo 模型的类 *ToDoItem*

```

package com.manning.gia.todo.model;

public class ToDoItem implements Comparable<ToDoItem> {
    private Long id;
    private String name;
    private boolean completed;
    (...)

}
  
```

模型持久化接口 *ToDoRepository*

```
package com.manning.gia.todo.repository;

import com.manning.gia.todo.model.ToDoItem;
import java.util.Collection;

public interface ToDoRepository {
    List<ToDoItem> findAll();
    ToDoItem findById(Long id);
    Long insert(ToDoItem toDoItem);
    void update(ToDoItem toDoItem);
    void delete(ToDoItem toDoItem);
}
```

接下来创建一个可扩展的、线程安全的实现：

```
package com.manning.gia.todo.repository;

public class InMemoryToDoRepository implements ToDoRepository {
    private AtomicLong currentId = new AtomicLong();
    private ConcurrentMap<Long, ToDoItem> toDos = new ConcurrentHashMap<Long, ToDoItem>()

    @Override
    public List<ToDoItem> findAll() {
        List<ToDoItem> ToDoItems = new ArrayList<ToDoItem>(toDos.values());
        Collections.sort(ToDoItems);
        return ToDoItems;
    }

    @Override
    public ToDoItem findById(Long id) {
        return toDos.get(id);
    }

    @Override
    public Long insert(ToDoItem ToDoItem) {
        Long id = currentId.incrementAndGet();
        ToDoItem.setId(id);
        toDos.putIfAbsent(id, ToDoItem);
        return id;
    }

    @Override
    public void update(ToDoItem ToDoItem) {
        toDos.replace(ToDoItem.getId(), ToDoItem);
    }

    @Override
    public void delete(ToDoItem ToDoItem) {
        toDos.remove(ToDoItem.getId());
    }
}
```

应用程序的入口

```
package com.manning.gia.todo;
import com.manning.gia.todo.utils.CommandLineInput;
import com.manning.gia.todo.utils.CommandLineInputHandler;

public class ToDoApp {
    public static final char DEFAULT_INPUT = '\u0000';
    public static void main(String args[]) {
        CommandLineInputHandler commandLineInputHandler = new
        CommandLineInputHandler();
        char command = DEFAULT_INPUT;

        while(CommandLineInput.EXIT.getShortCmd() != command) {
            commandLineInputHandler.printOptions();
            String input = commandLineInputHandler.readInput();
            char[] inputChars = input.length() == 1 ? input.toCharArray()
            char[] { DEFAULT_INPUT };
            command = inputChars[0];
            CommandLineInput commandLineInput = CommandLineInput.getCommandLineInputForIn
            commandLineInputHandler.processInput(commandLineInput);
        }
    }
}
```

到目前为止我们讨论了应用的组件和用户交互。接下来就要用Gradle实现项目的自动化构建，编译源代码、打包JAR文件、运行应用。

构建Java项目

上一节我们简要介绍了如何编写一个单机的To Do应用，接下来要打包部署成可执行的应用，我们需要编译源代码，生成的class文件需要打包到JAR文件中。JDK提供了javac 和jar工具帮助你实现这些任务，但是你也不想每次源代码发生变化时你都手动去执行这些任务吧。

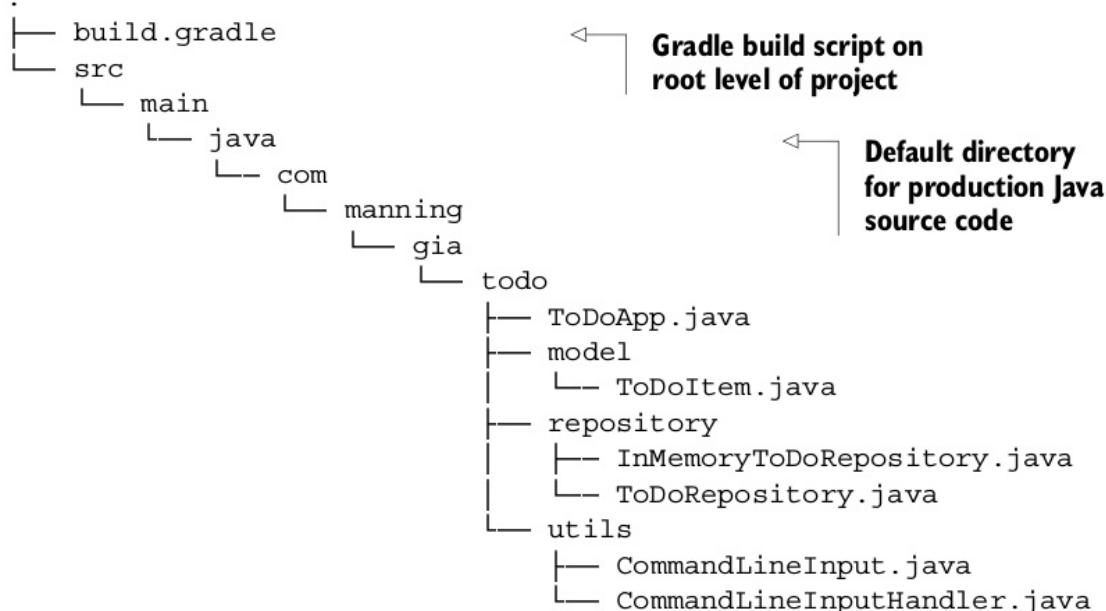
Gradle插件能够自动化完成这些任务，插件引入了一些领域特有的观念，其中一个Gradle插件就是java插件，Java插件不仅仅只有编译和打包的功能，它给你的项目安排了一个标准布局，并确保你所有的任务都是按序执行，现在该应用java插件来构建你的build脚本了。

使用java插件

每个Gradle项目都会创建一个build.gradle文件，如果你想使用java插件只需要添加下面这行代码：

```
apply plugin: 'java'
```

这一行代码足以构建你的项目，但是Gradle怎么知道你的源代码放在哪个位置呢？java插件的一个约定就是源代码的位置，默认情况下插件搜索src/main/java路径下的文件，你的包名com.manning.gia.todo会转换成源代码根目录下的子目录，创建build脚本之后你的项目结构应该是这样的：

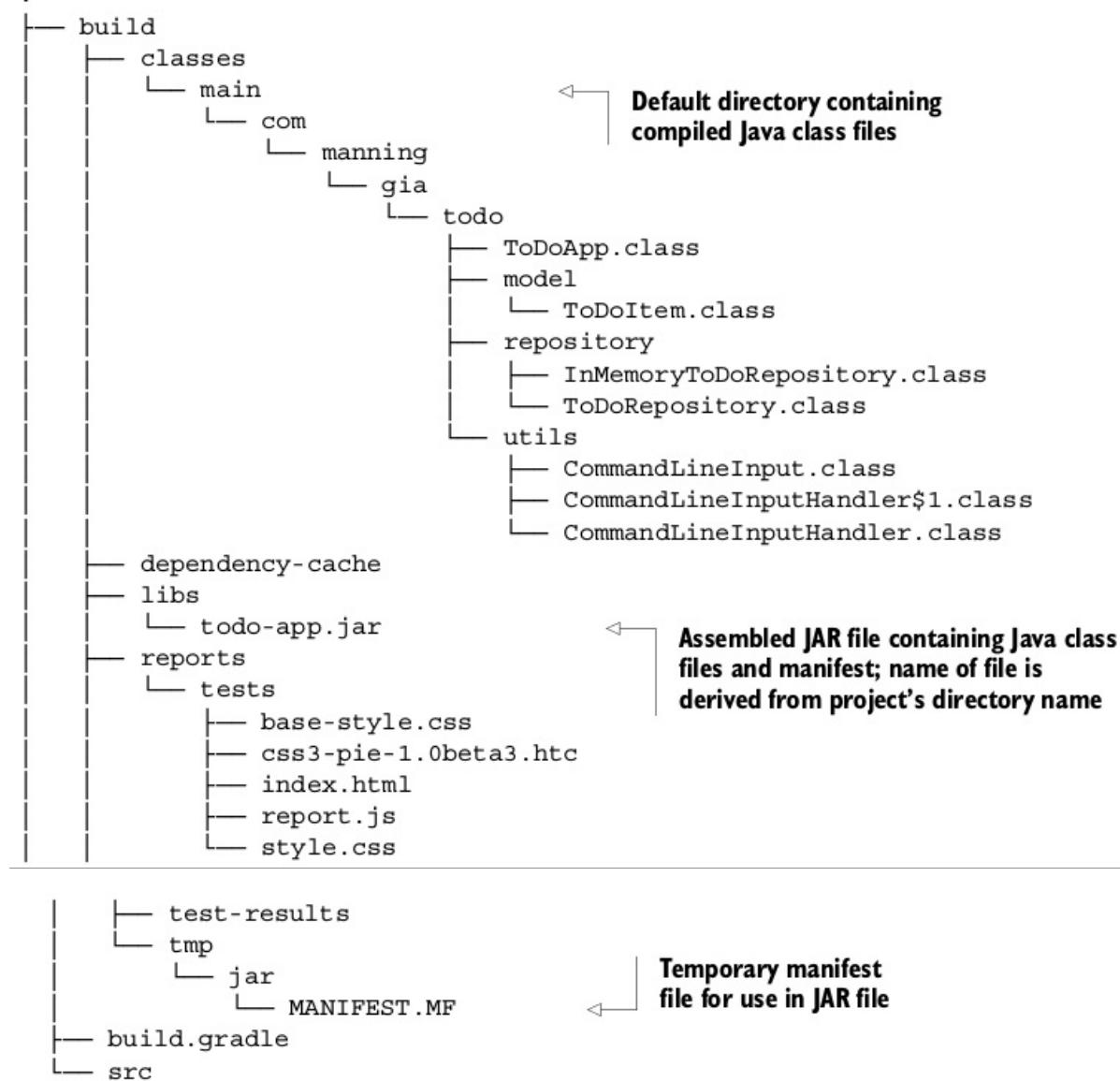


构建项目

现在你可以构建你的项目了，java插件添加了一个build任务到你项目中，build任务编译你的代码、运行测试然后打包成jar文件，所有都是按序执行的。运行gradle build之后你的输出应该是类似这样的：

```
$ gradle build
:compileJava
:processResources UP-TO-DATE
:classes
:jar
:assemble
:compileTestJava UP-TO-DATE
:processTestResources UP-TO-DATE
:testClasses UP-TO-DATE
:test
:check
:build
```

输出的每一行都表示一个可执行的任务，你可能注意到有一些任务标记为UP_TO-DATE,这意味着这些任务被跳过了，gradle能够自动检查哪些部分没有发生改变，就把这部分标记下来，省的重复执行。在大型的企业项目中可以节省不少时间。执行完gradle build之后项目结构应该是类似这样的：



在项目的根目录你可以找到一个build目录，这里包含了所有的输出，包含class文件，测试报告，打包的jar文件，以及一些用来归档的临时文件。如果你之前使用过maven,它的标准输出是target，这两个结构应该很类似。jar文件目录build/libs下可以直接运行，jar文件的名称直接由项目名称得来的，这里是todo-app。

运行项目

你只需要使用JDK的java命令就可以执行这个应用了：

```
$ java -cp build/classes/main com.manning.gia.todo.ToDoApp
--- To Do Application ---
Please make a choice:
(a)ll items
(f)ind a specific item
(i)nsert a new item
(u)pdate an existing item
(d)elete an existing item
(e)xit
>
```

接下来我们会学习如何自定义项目结构。

自定义你的项目

Java插件是一个非常固执的框架，对于项目很多的方面它都假定有默认值，比如项目布局，如果你看待世界的方法是不一样的，Gradle给你提供了一个自定义约定的选项。想知道哪些东西是可以配置的？可以参考这个手册：<http://www.gradle.org/docs/current/dsl/>，之前提到过，运行命令行gradle properties可以列出可配置的标准和插件属性以及他们的默认值。

修改你的项目和插件属性

接下来你将学习如何指定项目的版本号、Java源代码的兼容级别，前面你用的java命令来运行应用程序，你需要通过命令行选项-cp build/classes/main指定class文件的位置给Java运行时。但是要从JAR文件中启动应用，你需要在manifest文件MANIFEST.MF中包含首部Main-Class。看下面的脚本你就明白怎么操作了：

```
//Identifies project's version through a number scheme
version = 0.1

//Sets Java version compilation compatibility to 1.6
sourceCompatibility = 1.6

//Adds Main-Class header to JAR file's manifest

jar {
    manifest {
        attributes 'Main-Class': 'com.manning.gia.todo.ToDoApp'
    }
}
```

打包成JAR之后，你会发现JAR文件的名称变成了todo-app-0.1.jar，这个jar包含了main-class首部，你就可以通过命令java -jar build/libs/todo-app-0.1.jar运行了。

接下来学习如何改变项目的默认布局：

```
//Replaces conventional source code directory with list of different directories

sourceSets {
    main {
        java {
            srcDirs = ['src']
        }
    }
    //Replaces conventional test source code directory with list of different directories

    test {
        java {
            srcDirs = ['test']
        }
    }
}

//Changes project output property to directory out

buildDir = 'out'
```

配置和使用外部依赖

在Java世界里，依赖是分布的以JAR文件的形式存在，许多库都从仓库里获得，比如一个文件系统或中央服务器。Gradle需要你指定至少一个仓库作为依赖下载的地方，比如
mavenCentral : //Shortcut notation for configuring Maven Central 2 repository accessible under <http://repo1.maven.org/maven2>

```
repositories {
    mavenCentral()
}
```

定义依赖

接下来就是定义依赖，依赖通过group标识，name和version来确定，比如下面这个：

```
dependencies {
    compile group: 'org.apache.commons', name: 'commons-lang3', version: '3.1'
}
```

Gradle是通过配置来给依赖分组，Java插件引入的一个配置是compile，你可以很容易区分这个配置定义的依赖是用来编译源代码的。

解析依赖

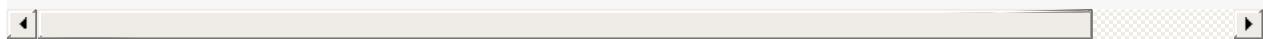
Gradle能够自动检测并下载项目定义的依赖：

```
$ gradle build
:compileJava
Download http://repo1.maven.org/maven2/org/apache/commons/commons-lang3/3.1/commons-lang3

Download http://repo1.maven.org/maven2/org/apache/commons/commons-parent/22/commons-paren

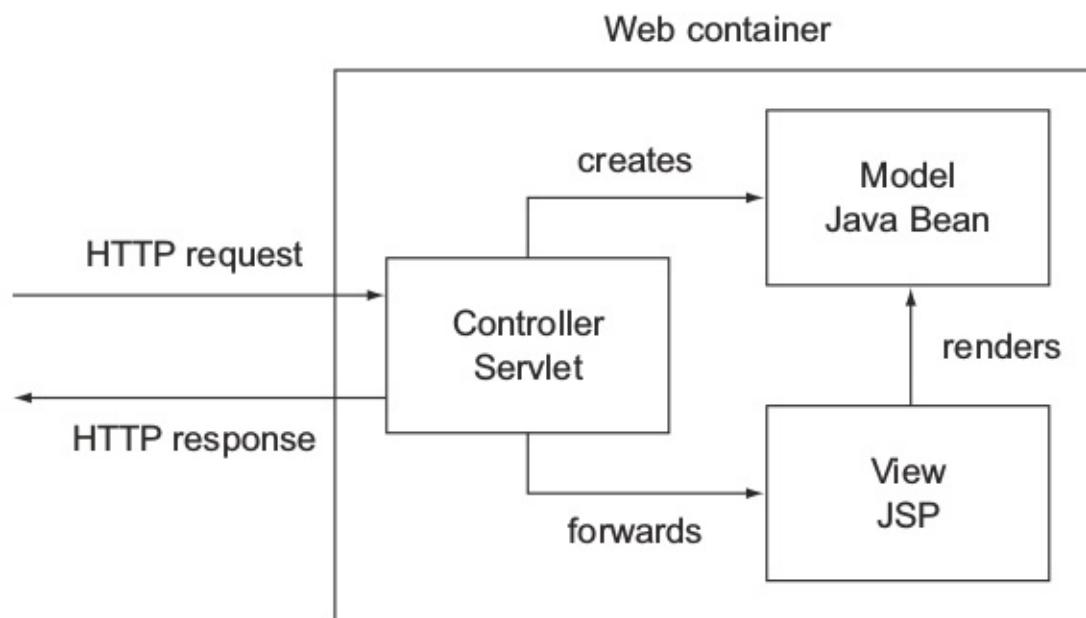
Download http://repo1.maven.org/maven2/org/apache/apache/9/apache-9.pom
Download http://repo1.maven.org/maven2/org/apache/commons/commons-lang3/3.1/commons-lang3

:processResources UP-TO-DATE
...
:build
```



用Gradle开发Web项目

Java服务端的Web组件(JavaEE)提供动态扩展能力允许你在web容器或者应用服务器中运行你的程序，就像Servlet这个名字的意思，接收客户端的请求返回响应，在MVC架构中充当控制器的角色，Servlet的响应通过视图组件--JSP来渲染，下图展示了一个典型的MVC架构的Java应用。

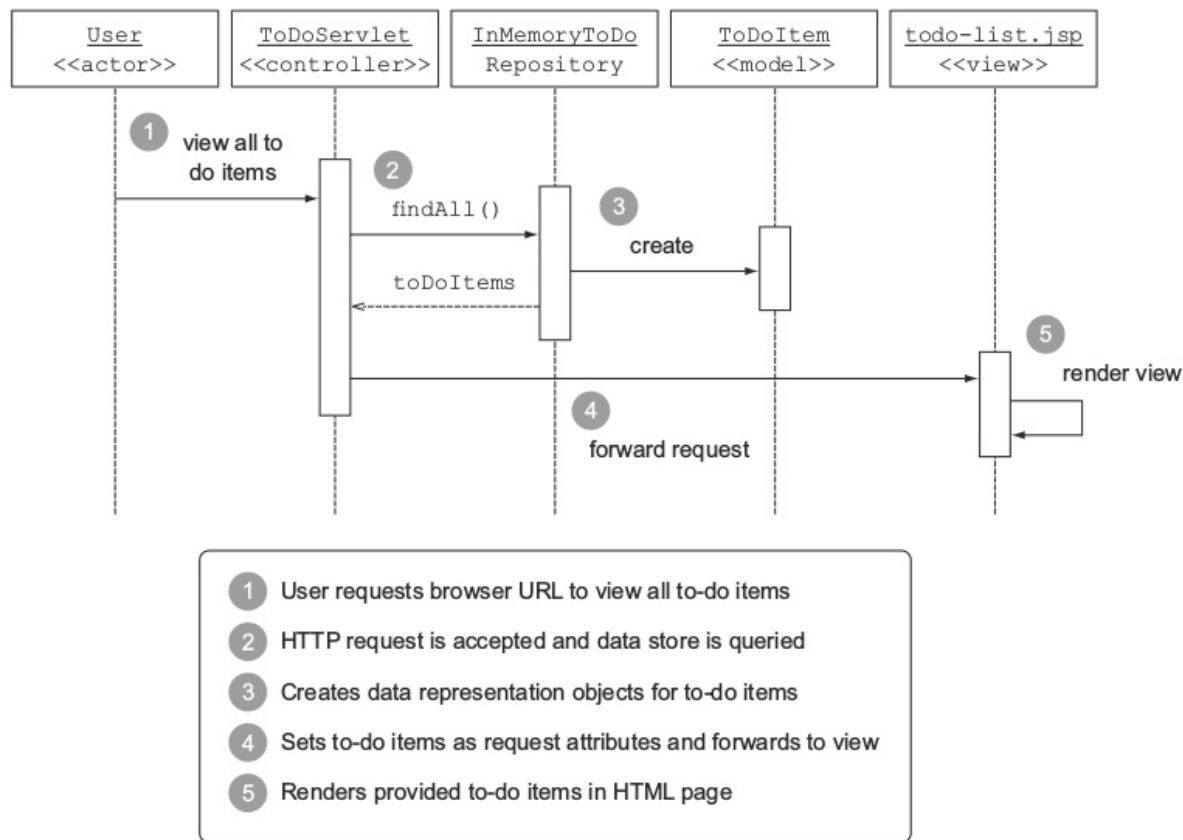


WAR(web application archive)用来捆绑Web组件、编译生成的class文件以及其他资源文件如部署描述符、HTML、JavaScript和CSS文件，这些文件组合在一起就形成了一个Web应用，要运行Java Web应用，WAR文件需要部署在一个服务器环境---Web容器。

Gradle提供拆箱插件用来打包WAR文件以及部署Web应用到本地的Servlet容器，接下来我们就来学习怎么把Java应用编程Web应用。

添加Web组件

接下来我们将创建一个Servlet，ToDoServlet，用来接收HTTP请求，执行CRUD操作，并将请求传递给JSP。你需要写一个todo-list.jsp文件，这个页面知道怎么去渲染todo items，提供一些UI组件比如按钮和指向CURD操作的链接，下图是用户检索和渲染todo items的流程：



Web 控制器

下面这个就是web 控制器ToDoServlet， 用来处理所有的URL请求：

```

package com.manning.gia.todo.web;

public class ToDoServlet extends HttpServlet {
    private ToDoRepository ToDoRepository = new InMemoryToDoRepository();

    @Override
    protected void service(HttpServletRequest request, HttpServletResponse response) throws
        String servletPath = request.getServletPath();
        String view = processRequest(servletPath, request);
        RequestDispatcher dispatcher = request.getRequestDispatcher(view);
        dispatcher.forward(request, response);

    }

    private String processRequest(String servletPath, HttpServletRequest request) {

        if(servletPath.equals("/all")) {
            List<ToDoItem> ToDoItems = ToDoRepository.findAll();
            request.setAttribute("ToDoItems", ToDoItems);
            return "/jsp/todo-list.jsp";
        }
        else if(servletPath.equals("/delete")) {
            ...
        }
        ...
        return "/all";
    }

}

```

对于每一个接收的请求，获取Servlet路径，基于CRUD操作在processRequest方法中处理请求，然后通过请求分派器请求传递给todo-list.jsp。

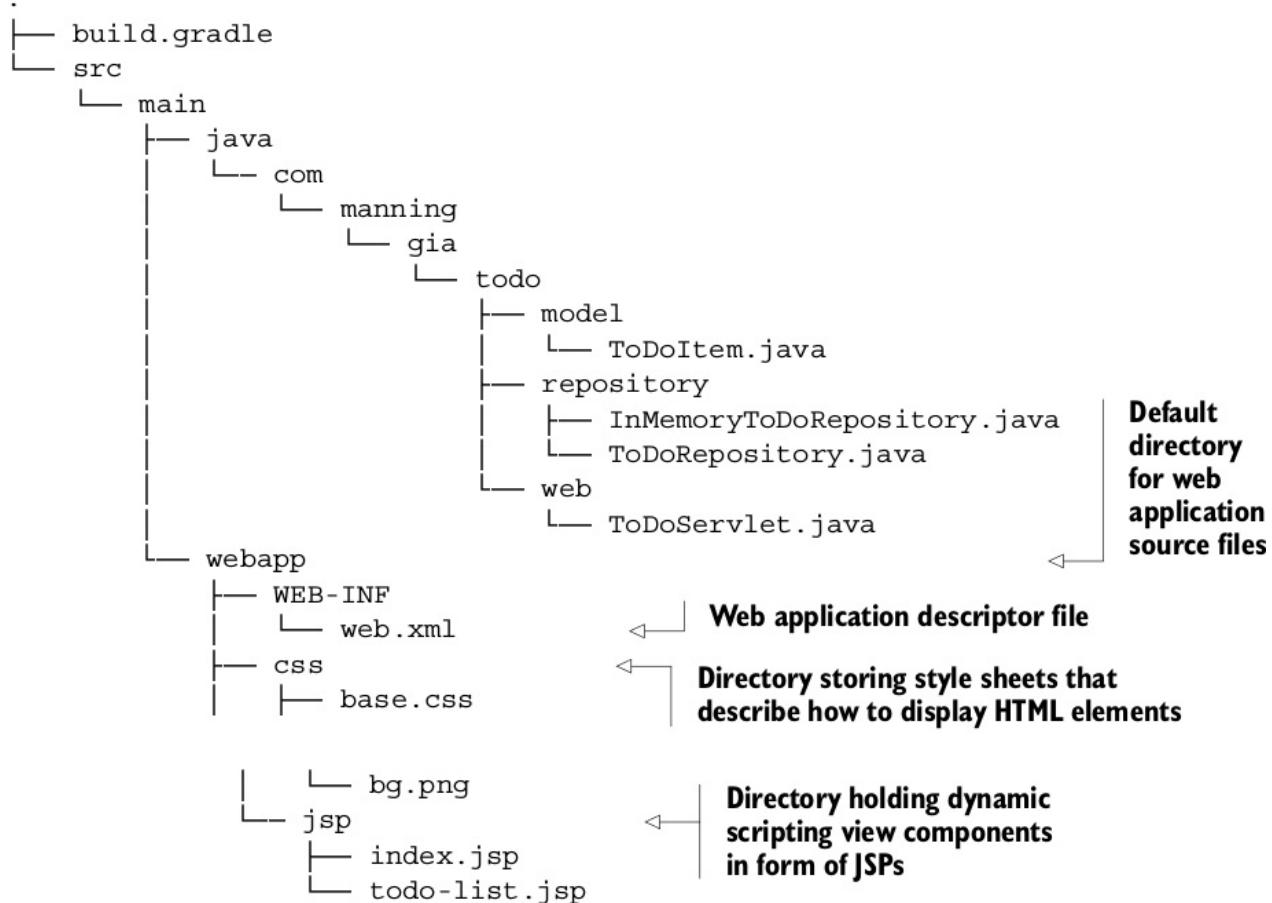
使用War和Jetty插件

Gradle支持构建和运行Web应用，接下来我将介绍两个web应用开发的插件War和Jetty，War插件继承了Java插件用来给web应用开发添加一些约定、支持打包War文件。Jetty是一个很受欢迎的轻量级的开源Web容器，Gradle的Jetty插件继承War插件，提供部署应用程序到嵌入的容器的任务。

既然War插件继承了Java插件，所有你在应用了War插件后无需再添加Java插件，即使你添加了也没有负面影响，应用插件是一个非幂等的操作，因此对于一个指定的插件只运行一次。

添加下面这句到你的build.gradle脚本中：apply plugin: 'war'

除了Java插件提供的约定外，你的项目现在多了Web应用的源代码目录，将打包成war文件而不是jar文件，Web应用默认的源代码目录是src/main/webapp,如果所有的源代码都在正确的位置，你的项目布局应该是类似这样的：



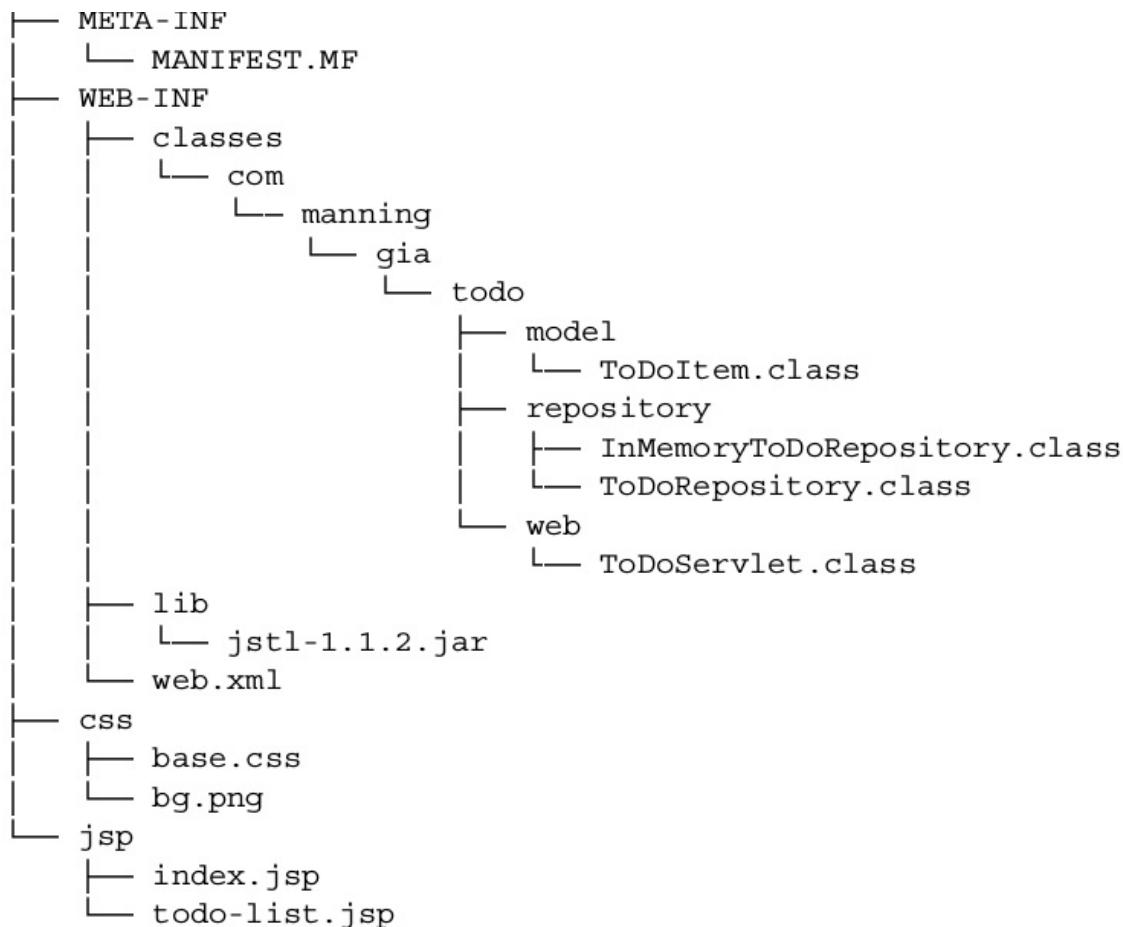
你用来实现Web应用的帮助类不是java标准的一部分，比如javax.servlet.HttpServlet，在运行build之前，你应该确保你声明了这些外部依赖，War插件引入了两个新的依赖配置，用于Servlet依赖的配置是providedCompile，这个用于那些编译器需要但是由运行时环境提供的依赖，你现在的运行时环境是Jetty，因此用provided标记的依赖不会打包到WAR文件里面，运行时依赖比如JSTL这些在编译器不需要，但是运行时需要，他们将成为WAR文件的一部分。

```
dependencies {
    providedCompile 'javax.servlet:servlet-api:2.5'
    runtime 'javax.servlet:jstl:1.1.2'
}
```

build Web项目和Java项目一样，运行gradle build后打包的WAR文件在目录build/libs下，输出如下：

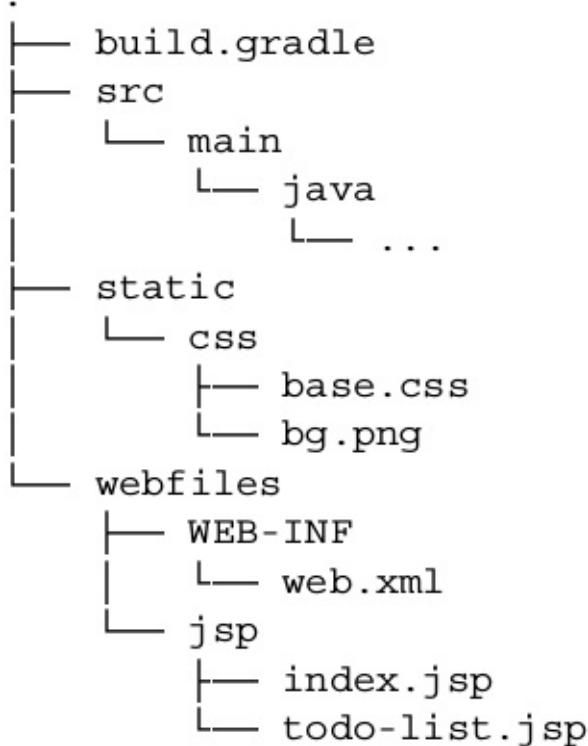
```
$ gradle build
:compileJava
:processResources UP-TO-DATE
:classes
:war
:assemble
:compileTestJava UP-TO-DATE
:processTestResources UP-TO-DATE
:testClasses UP-TO-DATE
:test
:check
:build
```

War插件确保打包的WAR文件符合JAVA EE规范，war任务拷贝 web应用源目录 src/main/webapp到WAR文件的根目录，编译的class文件保存在WEB-INF/classes,运行时依赖的库放在WEB-INF/libs,下面显示了WAR文件todo-webapp-0.1.war的目录结构：



自定义WAR插件

假设你把所有的静态文件放在static目录，所有的web组件放在webfiles，目录结构如下：



```

//Changes web application source directory

webAppDirName = 'webfiles'

//Adds directories css and jsp to root of WAR file archive
war {
from 'static'
}

```

在嵌入的**Web**容器中运行

嵌入式的Servlet容器完全不知到你的应用程序知道你提供准确的classpath和源代码目录，你可以手工编程提供，Jetty插件给你完成了所有的工作，你只需要添加下面一条命令：apply plugin: 'jetty'

运行Web应用需要用到的任务是jettyRun,启动Jetty容器并且无需创建WAR文件，这个命令的输出应该类似这样的：

```

$ gradle jettyRun
:compileJava
:processResources UP-TO-DATE
:classes
> Building > :jettyRun > Running at http://localhost:8080/todo-webapp-jetty

```

最后一行Jetty插件给你提供了一个URL用来监听HTTP请求，打开浏览器输入这个链接就能看到你编写的Web应用了。

Jetty插件默认监听的端口是8080，上下文路径是todo-webapp-jetty,你也可以自己配置成想要的：

```
jettyRun {  
    httpPort = 9090  
    contextPath = 'todo'  
}
```

这样你就把监听端口改成了9090,上下文目录改成了todo。

Gradle 包装器

你把你的Web应用给你的同伴Mike看，他看完之后觉得很有意思想加入你给项目添加一些高级特性。你把代码添加到版本控制系统当中（VCS）,因此它可以下载代码，由于Mike从来没有用过Gradle构建工具，所以他问你用的哪个版本的Gradle以及怎么安装Gradle，他也不知道怎么去配置Gradle，从以往的经验来看，Mike清醒的知道不同版本的构建工具或者运行环境对构建的影响有多大。对于在一个机器上可以运行，另一个机器无法运行的情况他看的太多了，经常是由于运行时环境不兼容的原因。

对于这个问题Gradle提供了一个非常方便和实用的方法：Gradle包装器，包装器是Gradle的一个核心特性，它允许你的机器不需要安装运行时就能运行Gradle脚本，而且她还能确保build脚本运行在指定版本的Gradle。它会从中央仓库中自动下载Gradle运行时，解压到你的文件系统，然后用来build。终极目标就是创建可靠的、可复用的、与操作系统、系统配置或Gradle版本无关的构建。

配置Gradle 包装器

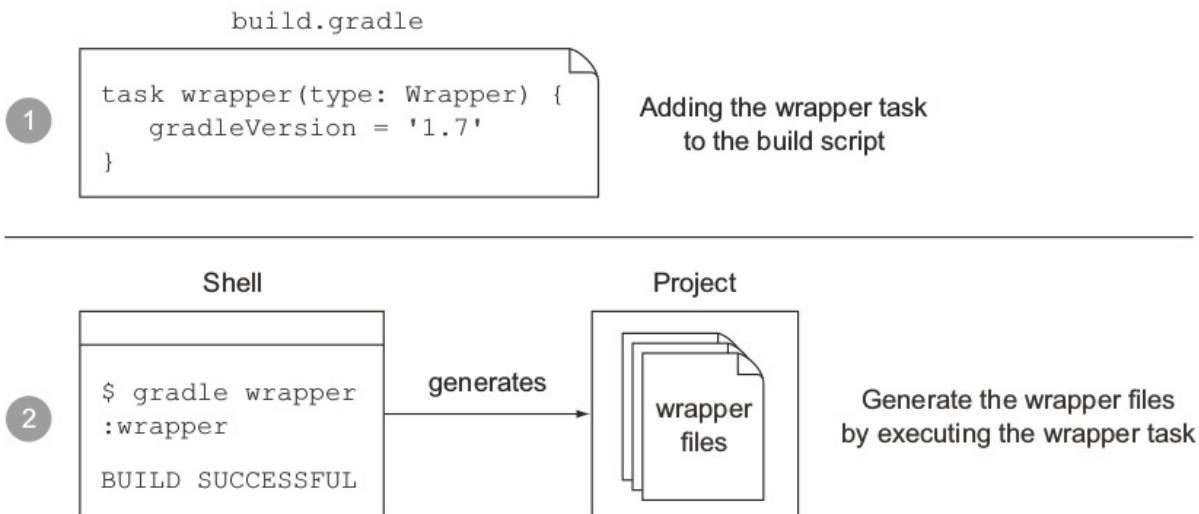
在设置你的包装器之前，你需要做两件事情：创建一个包装任务，执行这个任务生成包装文件。为了能让你的项目下载压缩的Gradle运行时，定义一个Wrapper类型的任务 在里面指定你想使用的Gradle版本：

```
task wrapper(type: Wrapper) {  
    gradleVersion = '1.7'  
}
```

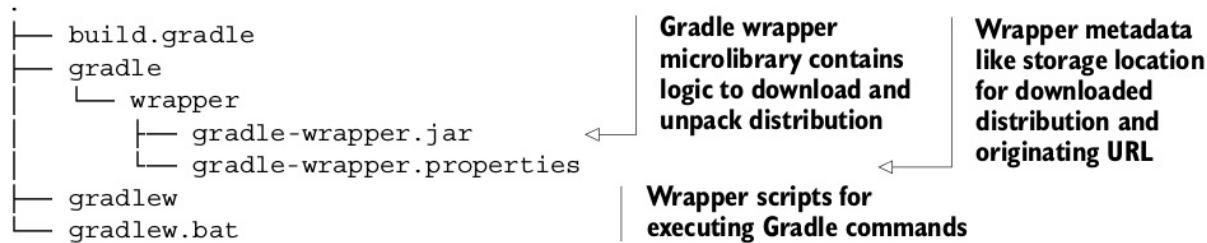
然后执行这个任务：

```
$ gradle wrapper  
:wrapper
```

整个过程如下图：



执行完之后，你就能看到下面这个wrapper文件和你的构建脚本：



记住你只需要运行gradle wrapper一次，以后你就能用wrapper来执行你的任务，下载下来的wrapper文件会被添加到版本控制系统中。如果你的系统中已经安装了Gradle运行时，你就不需要再添加一个gradle wrapper任务，你可以直接运行gradle wrapper任务，这个任务会使用你的Gradle当前版本来生成包装文件。

使用包装器

上面生成了两个执行脚本，一个是运行在*nix系统上的gradlew，另一个是运行在Windows系统上的gradlew.bat，你只需要根据你的系统环境来执行对应的那一个脚本，比如上面提到的Mike执行了gradlew.bat jettyRun任务，下面是输出：

```
> gradlew.bat jettyRun

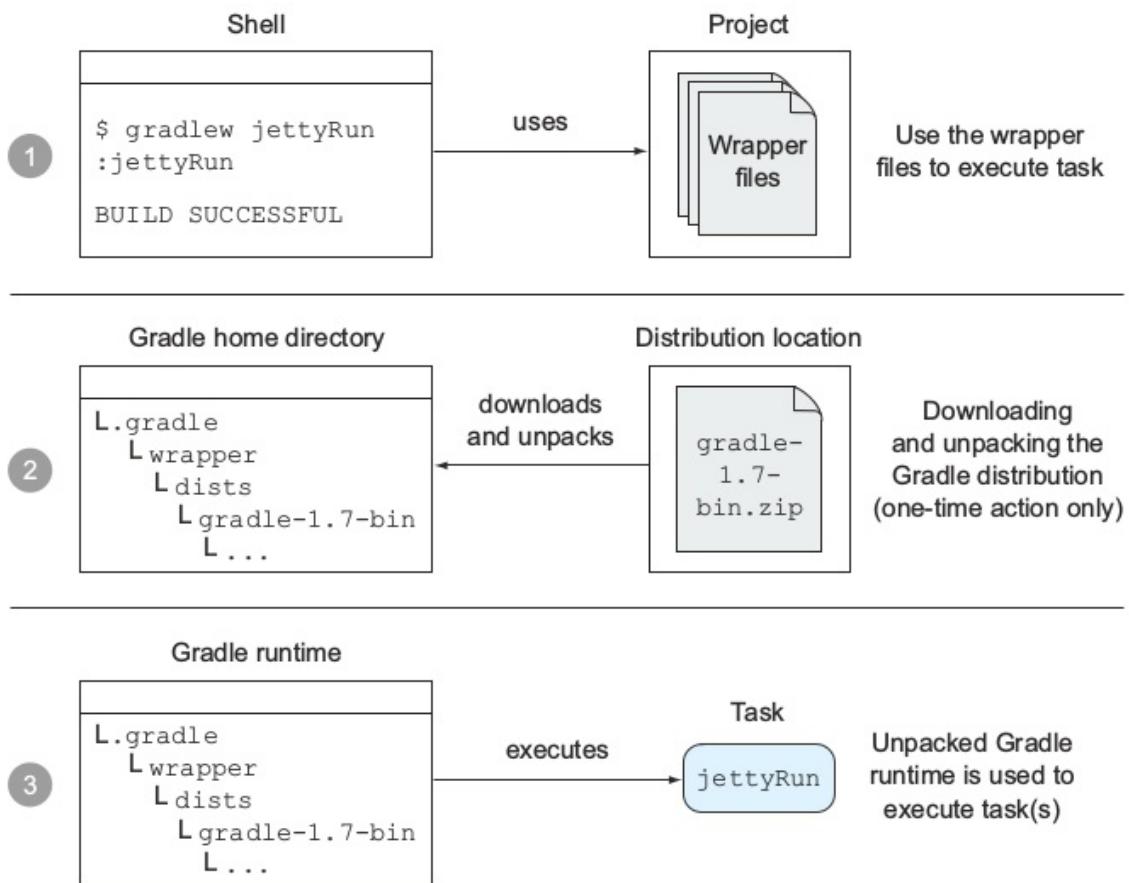
Downloading http://services.gradle.org/distributions/gradle-1.7-bin.zip
...
//Unzips compressed wrapper file to predefined local directory
Unzipping C:\Documents and Settings\Mike\.gradle\wrapper\dists\gradle-1.7- bin\35oej0jnbf

Set executable permissions for: C:\Documents and Settings\Mike\.gradle\wrapper\dists\gradle-1.7- bin\35oej0jnbf\gradle-1.7-bin\bin\gradlew.bat

:compileJava
:processResources UP-TO-DATE
:classes

> Building > :jettyRun > Running at http://localhost:9090/todo
```

整个过程如下：



自定义包装器

一些公司的安全措施非常严格，特别是当你给政府工作的时候，你能够访问外网的能力是被限制的，在这种情况下你怎么让你的项目使用Gradle包装器？所以你需要修改默认配置：

```
task wrapper(type: Wrapper) {  
    //Requested Gradle version  
    gradleVersion = '1.2'  
    //Target URL to retrieve Gradle wrapper distribution  
    distributionUrl = 'http://myenterprise.com/gradle/dists'  
    //Path where wrapper will be unzipped relative to Gradle home directory  
    distributionPath = 'gradle-dists'  
}
```

非常直接明显对不对？你还可以了解更多的特性，如果你想了解更多关于Gradle包装器DSL的信息，可以查看这个网

址：<http://gradle.org/docs/current/dsl/org.gradle.api.tasks.wrapper.Wrapper.html>

简介

在第三章，我们在Gradle核心插件的帮助下构建了一个Java Web项目，我们了解到这些插件都是可以自定义来适应自己的非标准化的构建需求、给你的项目添加可执行的构建逻辑来配置tasks。

在这一章，我们来学习Gradle构建的基本构建块(blocks)，比如项目和任务，以及他们是如何对应到Gradle API的类中，通过这些类的方法你可以获得一些属性来控制构建过程，你也将学习到如何使用属性来控制构建行为。

你将学习到如何定义简单的任务，更复杂一点的是编写自定义的任务类，接下来我们会接触到像访问任务属性、定义显式和隐式的依赖、添加递增的构建支持以及使用Gradle自带的任务类型。我们也会了解到Gradle的构建生命周期来更好的理解构建是怎么配置和执行的。

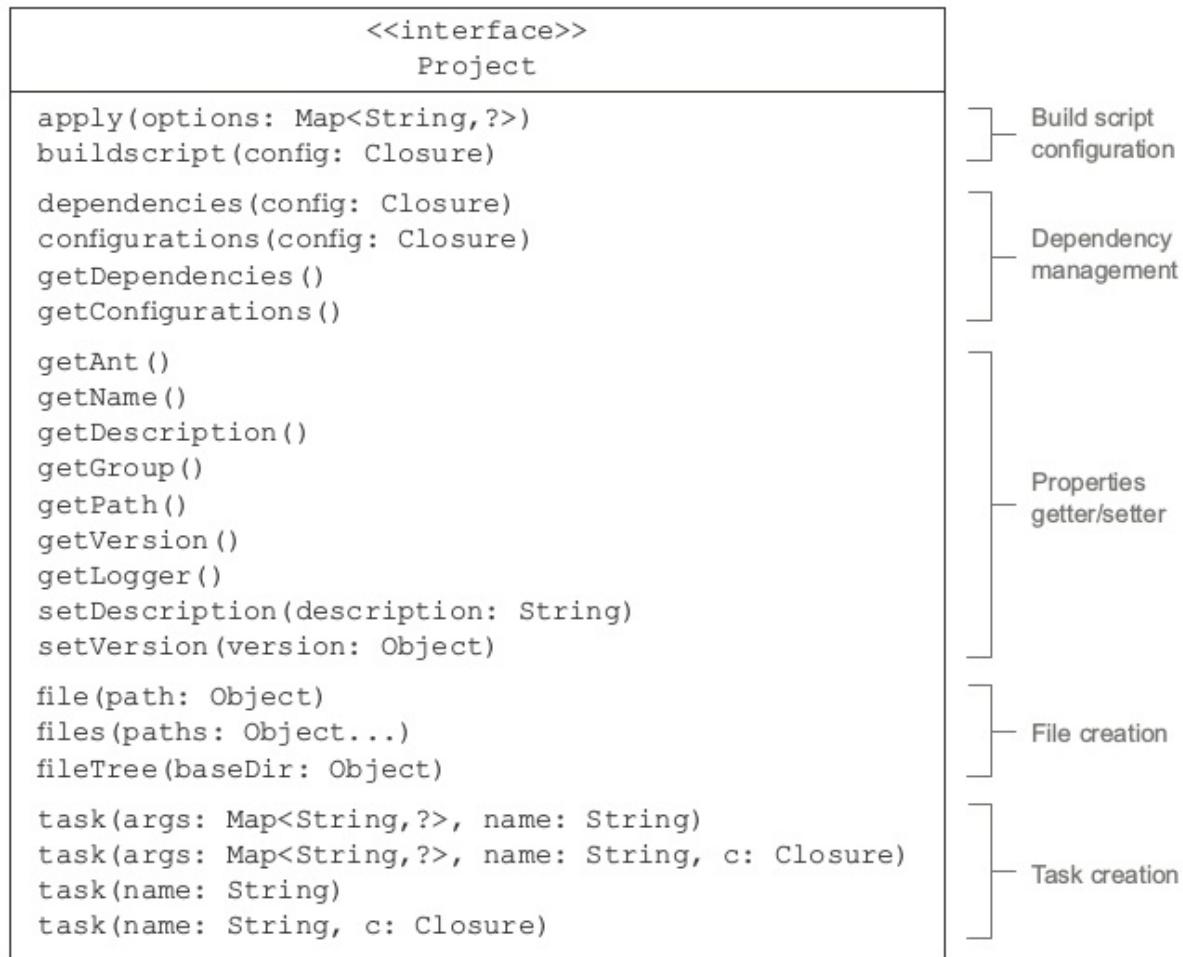
构建块

每个Gradle构建都包括三个基本的构建块：项目(projects)、任务(tasks)和属性(properties)，每个构建至少包括一个项目，项目包括一个或者多个任务，项目和任务都有很多个属性来控制构建过程。

Gradle运用了领域驱动的设计理念（DDD）来给自己的领域构建软件建模，因此Gradle的项目和任务都在Gradle的API中有一个直接的class来表示，接下来我们来深入了解每一个组件和它对应的API。

项目

在Gradle术语里项目表示你想构建的一个组件(比如一个JAR文件)，或者你想完成的一个目标(比如打包app)，如果你以前使用过Maven，你应该听过类似的概念。与Maven pom.xml相对应的是build.gradle文件，每个Gradle脚本至少定义了一个项目。当开始构建过程后，Gradle基于你的配置实例化org.gradle.api.Project这个类以及让这个项目通过project变量来隐式的获得。下图列出了API接口和最重要的方法。



一个项目可以创建新任务、添加依赖和配置、应用插件和其他脚本，许多属性比如name和description都是可以通过getter和setter方法来访问。

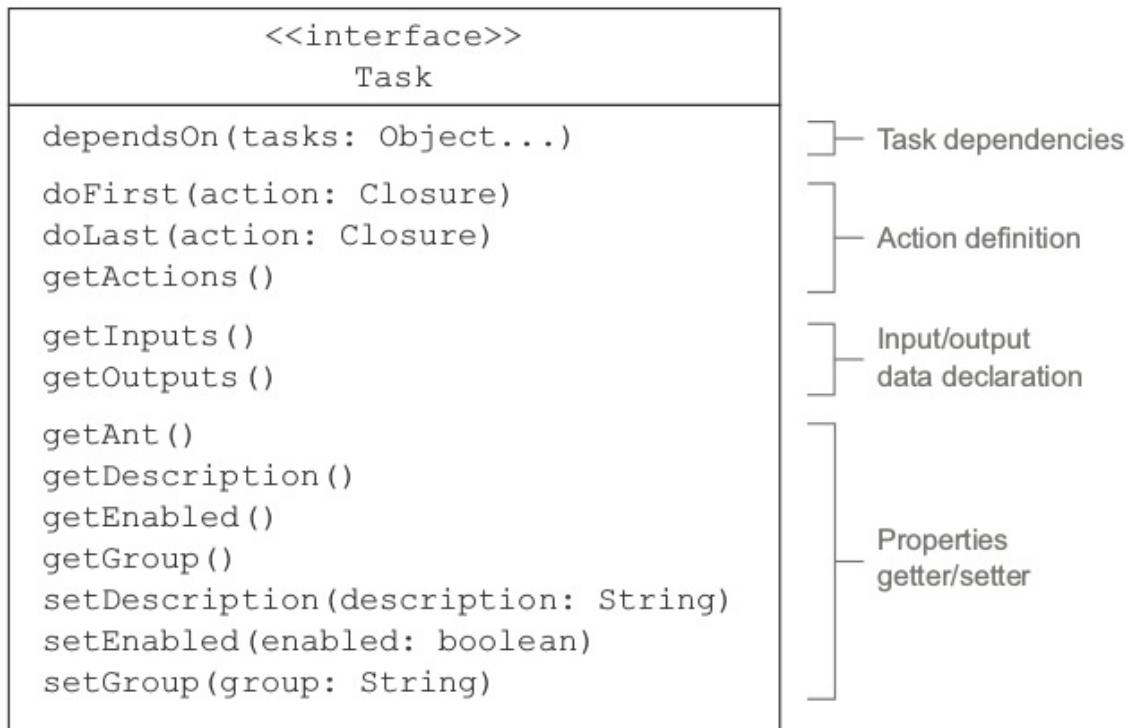
Project实例允许你访问你项目所有的Gradle特性，比如任务的创建和依赖了管理，记住一点当访问你项目的属性和方法时你并不需要显式的使用project变量--Gradle假定你的意思是Project实例，看看下面这个例子：//没有使用project变量来设置项目的描述
`setDescription("myProject")` //使用Groovy语法来访问名字和描述 `println "Description of project $name: " + project.description`

在之前的章节，我们只处理到单个peoject的构建，Gradle支持多项目的构建，软件设计一个很重要的概念是模块化，当一个软件系统变得越复杂，你越想把它分解成一个个功能性的模块，模块之间可以相互依赖，每个模块有自己的build.gradle脚本。

任务

我们在第二章的时候就创建了一些简单的任务，你应该了解两个概念：任务动作(actions)和任务依赖，一个动作就是任务执行的时候一个原子的工作，这可以简单到打印hello world,也可以复杂到编译源代码。很多时候一个任务需要在另一个任务之后执行，尤其是当一个任务的

输入依赖于另一个任务的输出时，比如项目打包成JAR文件之前先要编译成class文件，让我们来看看Gradle API中任务的表示：org.gradle.api.Task 接口。



属性

每个Project和Task实例都提供了setter和getter方法来访问属性，属性可以是任务的描述或者项目的版本号，在后续的章节，你会在具体例子中读取和修改这些属性值，有时候你要定义你自己的属性，比如，你想定义一个变量来引用你在构建脚本中多次使用的一个文件，Gradle允许你通过外部属性来定义自己的变量。

外部属性

外部属性一般存储在键值对中，要添加一个属性，你需要使用ext命名空间，看一个例子：

```
//Only initial declaration of extra property requires you to use ext namespace
project.ext.myProp = 'myValue' ext { someOtherProp = 123 }
```

```
//Using ext namespace to access extra property is optional
assert myProp == 'myValue'
println project.someOtherProp
ext.someOtherProp = 567
```

相似的，外部属性可以定义在一个属性文件中：通过在/.gradle路径或者项目根目录下的gradle.properties文件来定义属性可以直接注入到你的项目中，他们可以通过project实例来访问，注意/.gradle目录下只能有一个Gradle属性文件即使你有多个项目，在属性文件中定义的属性可以被所有的项目访问，假设你在你的gradle.properties文件中定义了下面的属性：

```
exampleProp = myValue  
someOtherProp = 455
```

你可以在项目中访问这两个变量：

```
assert project.exampleProp == 'myValue'  
  
task printGradleProperty << {  
    println "Second property: $someOtherProp"  
}
```

定义属性的其他方法

你也可以通过下面的方法来定义属性：

- 通过-P命令行选项来定义项目属性
- 通过-D命令行选项来定义系统属性
- 环境属性遵循这个模式：`ORG_GRADLE_PROJECT_propertyName=someValue`

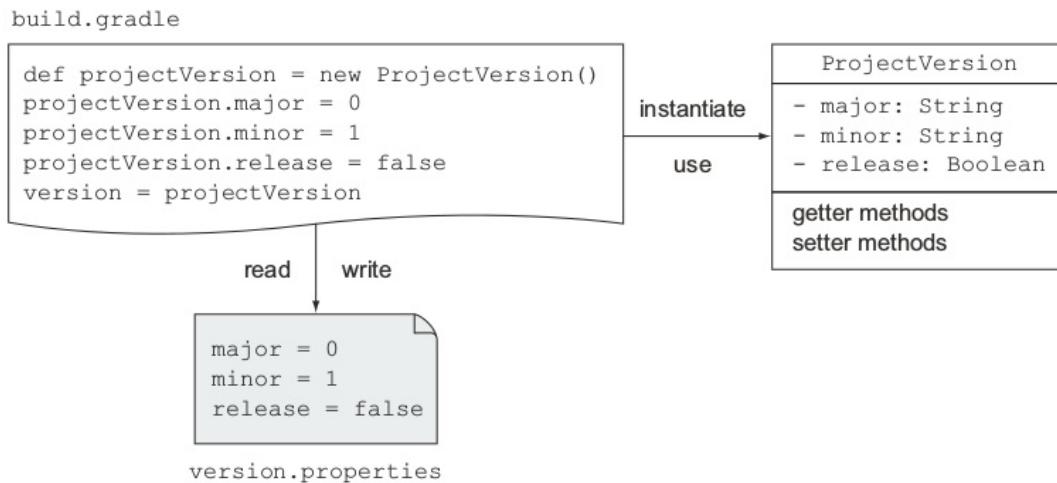
管理任务

每个新创建的任务都是org.gradle.api.DefaultTask类型，org.gradle.api.Task的标准实现，DefaultTask所有的域都是私有的，意味着他们只能通过setter和getter方法来访问，庆幸的是Groovy提供了一些语法糖来允许你通过名字来使用域。

管理项目的版本

许多公司或者开源组织有他们自己的发布版本的措施，一般用主版本号和次版本号来表示，也会用SNAPSHOT来表示项目正在开发中，版本号是通过String类型来表示，如果你想准确获得项目的主版本号，那应该怎么办？使用正则表达式匹配点号然后过滤得到主版本号和次版本号？如果我们用一个类来表示是不是更简单？

你可以很简单的通过类的域来设置、查询和修改你的版本号的某个部分，甚至你可以把版本信息直接保存在一个文件里，比如一个文件或者数据库里，避免通过修改构建脚本来更改版本号，如下图所示：



通过编程来控制版本对于自动化项目生命周期很有必要，比如：你的代码通过了单元测试准备交付了，当前的版本是1.3-SNAPSHOT，在打包成war文件之前你想把它变成发布版本1.3然后自动部署到服务器中，这些步骤可以划分为多个任务：一个用来修改项目的版本号另一个用于打包WAR文件。

声明任务的动作(actions)

动作就是在你的任务中放置构建逻辑的地方，Task接口提供了两个方法来声明任务的动作：`doFirst`和`doLast`，当任务执行的时候，定义在闭包里的动作逻辑就按顺序执行。

接下来我们会写一个简单的任务printVersion,任务的作用就是打印项目的版本号，在任务的最后一个动作定义这个逻辑。

```
version = '0.1-SNAPSHOT'

task printVersion {
    doLast {
        println "Version: $version"
    }
}
```

前面我们讲过左移操作符是方法doLast的快捷键，他们的作用是一样的，当你执行gradle printVersion,你应该得到下面的输出：

```
gradle printVersion
:printVersion
Version: 0.1-SNAPSHOT
```

如果你用doFirst方法的话输出的结果是一样的：

```
task printVersion {
    doFirst {
        println "Version: $version"
    }
}
```

给已经存在的任务添加动作

到目前为止，你只是给printVersion这个任务添加了单个动作，要么是第一个或者最后一个，对于每个任务可以有多个动作，实际上，当任务创建的时候你可以添加任意多个动作，每一个任务都有一个动作清单，他们在运行的时候是执行的，接下来我们来修改之前的例子：

```
task printVersion {
    //任务的初始声明可以添加first和last动作
    doFirst {
        println "Before reading the project version"
    }

    doLast {
        println "Version: $version"
    }
}
```

//你可以在任务的动作列表的最前面添加其他任务，比如：

```
printVersion.doFirst { println "First action" }
```

由此可知，我们可以添加额外的动作给已经存在的任务，当你想添加动作的那个任务不是你自己写的时候这会非常有用，你可以添加一些自定义的逻辑，比如你可以添加doFirst动作到compile-Java任务来检查项目是否包含至少一个source文件。

访问任务属性

接下来我们来改善一下输出版本号的方法，Gradle提供一个基于SLF4J库的日志实现，除了实现了基本的日志级别（DEBUG, ERROR, INFO, TRACE, WARN）外，还添加了额外的级别，日志实例可以通过任务的方法来直接访问，接下来，你将用QUIET级别打印项目的版本号：

```
task printVersion << {
    logger.quiet "Version: $version"
}
```

访问任务的属性是不是很容易？接下来我将给你展示两个其他的属性，group和description，两个都是documentation任务的一部分，description属性简短的表示任务的目的，group表示任务的逻辑分组。

```
task printVersion(group: 'versioning', description:      'Prints project version.') << {
    logger.quiet "Version: $version"
}
```

你也可以通过setter方法来设置属性：

```
task printVersion {
    group = 'versioning'
    description = 'Prints project version.'
    doLast {
        logger.quiet "Version: $version"
    }
}
```

当你运行gradle tasks,你会看到任务显示在正确的分组里和它的描述信息：

```

gradle tasks
:tasks
...
Versioning tasks
-----
printVersion - Prints project version.
...

```

定义任务依赖

`dependsOn`方法用来声明一个任务依赖于一个或者多个任务，接下来通过一个例子来讲解运用不同的方法来应用依赖：

```

task first << { println "first" }
task second << { println "second" }

//声明多个依赖
task printVersion(dependsOn: [second, first]) << {
    logger.quiet "Version: $version"
}

task third << { println "third" }
//通过任务名称来声明依赖
third.dependsOn('printVersion')

```

你可以通过命令行调用`third`任务来执行这个任务依赖链：

```

$ gradle -q third
first
second
Version: 0.1-SNAPSHOT
third

```

仔细看这个执行顺序，你有没有发现`printVersion`声明了对`second`和`first`任务的依赖，但是`first`在`second`任务前执行了，Gradle里面任务的执行顺序并不是确定的。

任务依赖执行顺序

Gradle并不保证依赖的任务能够按顺序执行，`dependsOn`方法只是定义这些任务应该在这个任务之前执行，但是这些依赖的任务具体怎么执行它并不关心，如果你习惯用命令式的构建工具来定义依赖（比如ant）这可能会难以理解。在Gradle里面，执行顺序是由任务的输入输出特性决定的，这样做有很多优点，比如你想修改构建逻辑的时候你不需要去了解整个任务依赖链，另一方面，因为任务不是顺序执行的，就可以并发的执行来提高性能。

终结者任务

在实际情况中，你可能需要在一个任务执行之后进行一些清理工作，一个典型的例子就是Web容器在部署应用之后要进行集成测试，Gradle提供了一个finalizer任务来实现这个功能，你可以用finalizedBy方法来结束一个指定的任务：

```
task first << { println "first" }
task second << { println "second" }
//声明first结束后执行second任务
first.finalizedBy second
```

你会发现任务first结束后自动触发任务second：

```
$ gradle -q first
first
second
```

添加随意的代码

接下来我们来学习怎么在build脚本中定义一些随机的代码，在实际情况下，如果你熟悉Groovy的语法你可以编写一些类或者方法，接下来你将会创建一个表示版本的类，在Java中一个class遵循bean的约定（POJO），就是添加setter和getter方法来访问类的域，到后面发现手工写这些方法很烦人，Groovy有个对应的概念叫POGO(plain-old Groovy object)，他们的setter和getter方法在生成字节码的时候自动添加，因此运行的时候可以直接访问，看下面这个例子：

```

version = new ProjectVersion(0, 1)

class ProjectVersion {
    Integer major
    Integer minor
    Boolean release

    ProjectVersion(Integer major, Integer minor) {
        this.major = major
        this.minor = minor
        this.release = Boolean.FALSE
    }

    ProjectVersion(Integer major, Integer minor,      Boolean release) {
        this(major, minor)
        this.release = release
    }

    @Override
    String toString() {
        //只有release为false的时候才添加后缀SNAPSHOT
        "$major.$minor${release ? '' : '-SNAPSHOT'}"
    }
}

```

当运行这个修改的脚本之后，你可以看到printVersion的输出和之前一样，但是你还是得手工修改build脚本来更改版本号，接下来你将学习如何把版本号存储在一个文件里然后配置你的脚本去读取这个配置。

任务的配置

在你写代码之前，你要新建一个属性文件version.properties,内容如下：

```

major = 0
minor = 1
release = false

```

添加任务配置块

接下来我们将声明一个任务loadVersion来从属性文件中读取版本号并赋给ProjectVersion实例，第一眼看起来和其他定义的任务一样，仔细一看你会注意到你没有定义动作或者使用左移操作符，在Gradle里称之为任务配置块(task configuration)。

```

ext.versionFile = file('version.properties')
//配置任务没有左移操作符
task loadVersion {
    project.version = readVersion()
}

ProjectVersion readVersion() {
    logger.quiet 'Reading the version file.'
    //如果文件不存在抛出异常
    if(!versionFile.exists()) {
        throw new GradleException("Required version file does not exist:$versionFile.canon")
    }

Properties versionProps = new Properties()

//groovy的file实现了添加方法通过新创建的流来读取

versionFile.withInputStream { stream ->
versionProps.load(stream)
}
//在Groovy中如果这是最后一个语句你可以省略return关键字
new ProjectVersion(versionProps.major.toInteger(),
    versionProps.minor.toInteger(), versionProps.release.toBoolean())
}

```

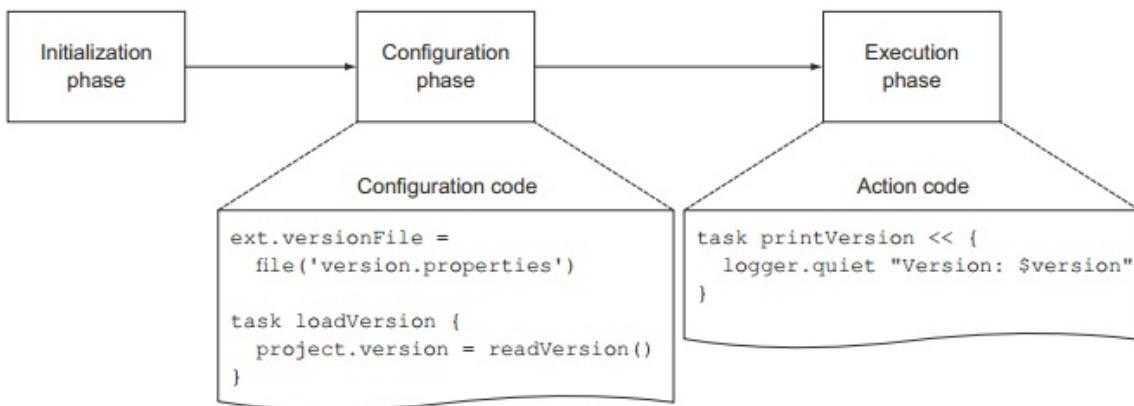
接下来运行printVersion，你会看到loadVersion任务先执行了：

```

$ gradle printVersion
Reading the version file.
:printVersion
Version: 0.1-SNAPSHOT

```

你也许会很奇怪这个任务是怎么调用的，你没有声明依赖，也没有在命令行中调用它。任务配置块总是在任务动作之前执行的，理解这个行为的关键就是Gradle的构建生命周期，我们来看下Gradle的构建阶段：



Gradle的构建生命周期

无论你什么时候执行一个gradle build,都会经过三个不同的阶段：初始化、配置和执行。

在初始化阶段，Gradle给你的项目创建一个Project实例，你的构建脚本只定义了单个项目，在多项目构建的上下文环境中，构建的阶段更为重要。根据你正在执行的项目，Gradle找出这个项目的依赖。

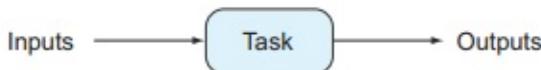
下一个阶段就是配置阶段，Gradle构建一些在构建过程中需要的一些模型数据，当你的项目或者指定的任务需要一些配置的时候这个阶段很有帮助。

记住不管你执行哪个build哪怕是gradle tasks配置代码都会执行

在执行阶段任务按顺序执行，执行顺序是通过依赖关系决定的，标记为up-to-date的任务会跳过，比如任务B依赖于任务A，当你运行gradle B的时候执行顺序将是A->B。

声明任务的输入和输出

Gradle通过比较两次build之间输入和输出有没有变化来确定这个任务是否是最新的，如果从上一个执行之后这个任务的输入和输出没有发生改变这个任务就标记为up-to-date，跳过这个任务。



输入可以是一个目录、一个或者多个文件或者随机的属性，任务的输出可以是路径或者文件，输入和输出在DefaultTask类中用域来表示。假设你想创建一个任务把项目的版本由SNAPSHOT改为release，下面的代码定义一个新任务给release变量赋值为true，然后把改变写入到文件中。

```

task makeReleaseVersion(group: 'versioning', description: 'Makes project a release version')
version.release = true
//ant的propertyfile任务提供很方便的方法来修改属性文件
ant.propertyfile(file: versionFile) {
    entry(key: 'release', type:'string',operation: '=', value: 'true')
}
  
```

运行这个任务会修改版本属性并写入到文件中。

```
$ gradle makeReleaseVersion
:makeReleaseVersion

$ gradle printVersion
:printVersion
Version: 0.1
```

编写自定义的任务

`makeReleaseVersion`的逻辑比较简单，你可能不用考虑代码维护的问题，随着构建逻辑越来越复杂，你添加了越来越多的简单的任务，这时候你就有需要用类和方法来结构化你的代码，你可以把你编写源代码的那一套代码实践搬过来。

编写自定义任务类

之前提到过，Gradle会给每一个任务创建一个`DefaultTask`类型的实例，当你要创建一个自定义的任务时，你需要创建一个继承自`DefaultTask`的类，看看下面这个例子：

```
class ReleaseVersionTask extends DefaultTask {
    //通过注解声明任务的输入和输出
    @Input Boolean release
    @OutputFile File destFile

    ReleaseVersionTask() {
        //在构造器里设置任务的分组和描述
        group = 'versioning'
        description = 'Makes project a release version.'
    }
    //通过注解声明要执行的任务
    @TaskAction
    void start() {
        project.version.release = true
        ant.propertyfile(file: destFile) {
            entry(key: 'release', type: 'string', operation: '=', value: 'true')
        }
    }
}
```

通过注解来表达输入和输出

任务输入和输出注解给你的实现添加了语法糖，他们和调用`TasksInputs`和`TaskOutputs`方法是一样的效果，你一眼就知道任务期望什么样的输入数据以及会产生什么输出。我们使用`@Input`注解来声明输入属性`release`，用`@OutputFile`来定义输出文件。

使用自定义的任务

上面我们实现了自定义的动作方法，但是我们怎么使用这个方法，你需要在build脚本中创建一个ReleaseVersionTask类型的任务，通过给属性赋值来设定输入和输出：

```
//定义一个ReleaseVersionTask类型的任务
task makeReleaseVersion(type: ReleaseVersionTask) {
    //设定任务属性
    release = version.release
    destFile = versionFile
}
```

复用自定义的任务

假设你在另一个项目中想使用前面这个自定义的任务，在另一个项目中需求又不太一样，用来表示版本的POGO有不同的域，比如下面这个：

```
class ProjectVersion {
    Integer min
    Integer maj
    Boolean prodReady

    @Override
    String toString() {
        "$maj.$min${prodReady? '' : '-SNAPSHOT'}"
    }
}
```

此外，你还想把版本文件名改为project-version.properties，需要怎么做才能复用上面那个自定义的任务呢？

```
task makeReleaseVersion(type: ReleaseVersionTask) {
    release = version.prodReady
    //不同的版本文件
    destFile = file('project-version.properties')
}
```

Gradle自带的任务类型

Gradle自带的任务类型继承自DefaultTask，Gradle提供了很多自带的任务类型，这里我只介绍两个，Zip和copy用在发布项目中。

//eg. 使用任务类型来备份发布版本

```

task createDistribution(type: Zip, dependsOn:      makeReleaseVersion) {
    //引用war任务的输出
    from war.outputs.files
    //把所有文件放进ZIP文件的src目录
    from(sourceSets*.allSource) {
        into 'src'
    }
    //添加版本文件
    from(rootDir) {
        include versionFile.name
    }
}

task backupReleaseDistribution(type: Copy) {
    //引用createDistribution的输出
    from createDistribution.outputs.files
    into "$buildDir/backup"
}

task release(dependsOn: backupReleaseDistribution)      << {
    logger.quiet 'Releasing the project...'
}

```

任务依赖推导

你可能注意到上面通过`dependsOn`方法来显示声明两个任务之间的依赖，可是，一些任务并不是直接依赖于其他任务(比如上面`createDistribution`依赖于`war`)。Gradle怎么知道在任务之前执行哪个任务？通过使用一个任务的输出作为另一个任务的输入，依赖就推导出来了，结果依赖的任务自动执行了，我们来看一下完整的执行图：

```

$ gradle release
:makeReleaseVersion
:compileJava
:processResources UP-TO-DATE
:classes
:war
:createDistribution
:backupReleaseDistribution
:release
Releasing the project...

```

运行`build`之后你可以在`build/distribution`目录找到生成的ZIP文件，这是打包任务的默认输出目录，下面这个图是生成的目录树：

```

.
├── build
│   ├── backup
│   │   └── todo-webapp-0.1.zip
│   ├── distributions
│   │   └── todo-webapp-0.1.zip
│   └── libs
│       └── todo-webapp-0.1.war
├── build.gradle
└── src
    └── version.properties

```

在buildSrc目录创建代码

在前面我们创建了两个类，ProjectVersion和ReleaseVersionTask，这些类可以移动到你项目的buildSrc目录，buildSrc目录是一个放置源代码的可选目录，你可以很容易的管理你的代码。Gradle采用了标准的项目布局，java代码在src/main/java目录，Groovy代码应该在src/main/groovy目录，在这些目录的任何代码都会自动编译然后放置到项目的classpath目录。这里你是在处理class，你可以把他们放到指定的包里面，假如com.manning.gia,下面显示了Groovy类在项目中的目录结构：

```

.
├── build.gradle
├── buildSrc
│   └── src
│       └── main
│           └── groovy
│               └── com
│                   └── manning
│                       └── gia
│                           ├── ProjectVersion.groovy
│                           └── ReleaseVersionTask.groovy
└── src
    └── ...
└── version.properties

```

不过要记住把这些类放在源代码目录需要额外的工作，这和在脚本文件中定义有点不一样，你需要导入Gradle的API，看看下面这个例子：

```

package com.manning.gia
import org.gradle.api.DefaultTask
import org.gradle.api.tasks.Input
import org.gradle.api.tasks.OutputFile
import org.gradle.api.tasks.TaskAction

class ReleaseVersionTask extends DefaultTask {
    (...)

}

```

反过来，你的构建脚本需要从buildSrc中导入编译的classes(比如 com.manning.gia.ReleaseVersionTask)，下面这个是编译任务输出：

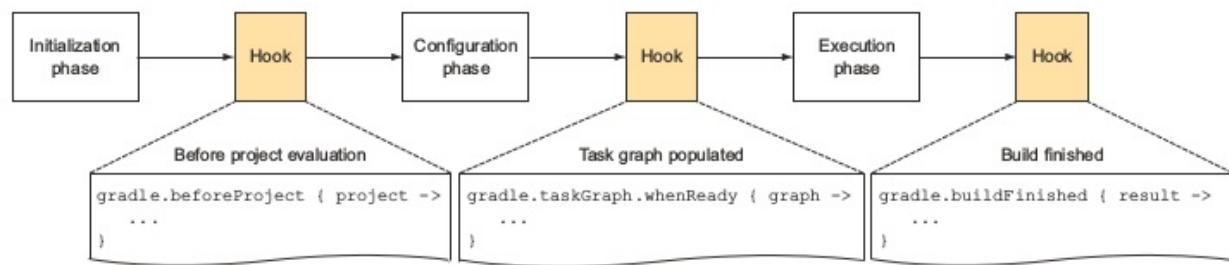
```
$ gradle makeReleaseVersion
:buildSrc:compileJava UP-TO-DATE
:buildSrc:compileGroovy
:buildSrc:processResources UP-TO-DATE
:buildSrc:classes
:buildSrc:jar
:buildSrc:assemble
:buildSrc:compileTestJava UP-TO-DATE
:buildSrc:compileTestGroovy UP-TO-DATE
:buildSrc:processTestResources UP-TO-DATE
:buildSrc:testClasses UP-TO-DATE
:buildSrc:test
:buildSrc:check
:buildSrc:build
:makeReleaseVersion UP-TO-DATE
```

到此为止你学习了简单任务的创建，自定义的task类，指定Gradle API提供的task类型，查看了任务动作和任务配置的区别，以及他们的使用情形，任务配置和任务动作是在不同阶段执行的。

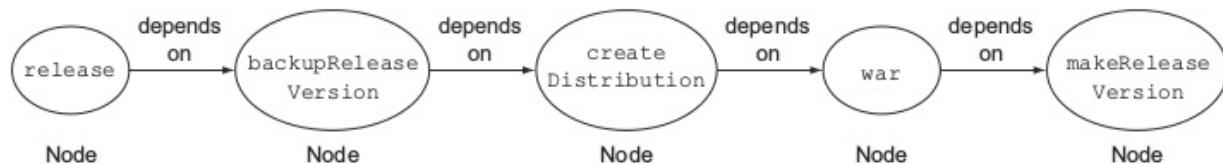
掌握构建生命周期

作为一个构建脚本的开发者，你不应该局限于编写任务动作或者配置逻辑，有时候你想在指定的生命周期事件发生的时候执行一段代码。生命周期事件可以在指定的生命周期之前、之中或者之后发生，在执行阶段之后发生的生命周期事件就该是构建的完成了。

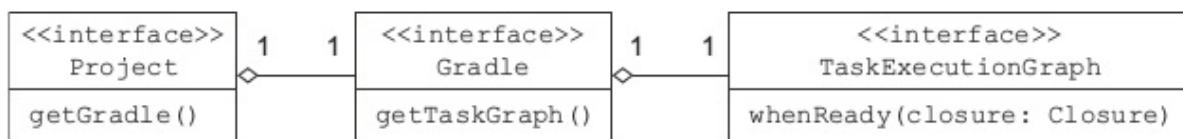
假设你希望在构建失败时能够在开发阶段尽早得到反馈，给构建生命周期事件添加回调有两种方法：一是通过闭包，二是实现Gradle API的一个监听接口，Gradle并没有要求你监听生命周期事件，这完全决定于你，通过监听器实现的优势就是可以给你的类写单元测试，看看下面这幅图会有一点直观的印象：



在配置阶段，Gradle决定在任务在执行阶段的执行顺序，依赖关系的内部结构是通过直接的无环图(DAG)来表示的，图中的每一个任务称为一个节点，每一个节点通过边来连接，你很有可能通过dependsOn或者隐式的依赖推导来创建依赖关系。记住DAG图从来不会有环，就是说一个已经执行的任务不会再次执行，下面这幅图将要的展示了这个过程：



回想一下之前我们实现的makeReleaseVersion任务是在release任务之前执行的，我们可以编写一个生命周期回调方法来取代之前写一个任务来执行版本修改任务。构建系统准确知道在执行之前应该运行哪些任务，你可以查询任务图来查看它是否存在，下面这幅图展示了访问任务执行图的相关接口：



接下来我们来添加相应的监听方法，下面这段代码通过调用whenReady方法来注册回调接口，当任务图创建的时候这个回调会自动执行，你知道这个逻辑会在任何任务之前执行，所以你可以移除makeReleaseVersion任务。

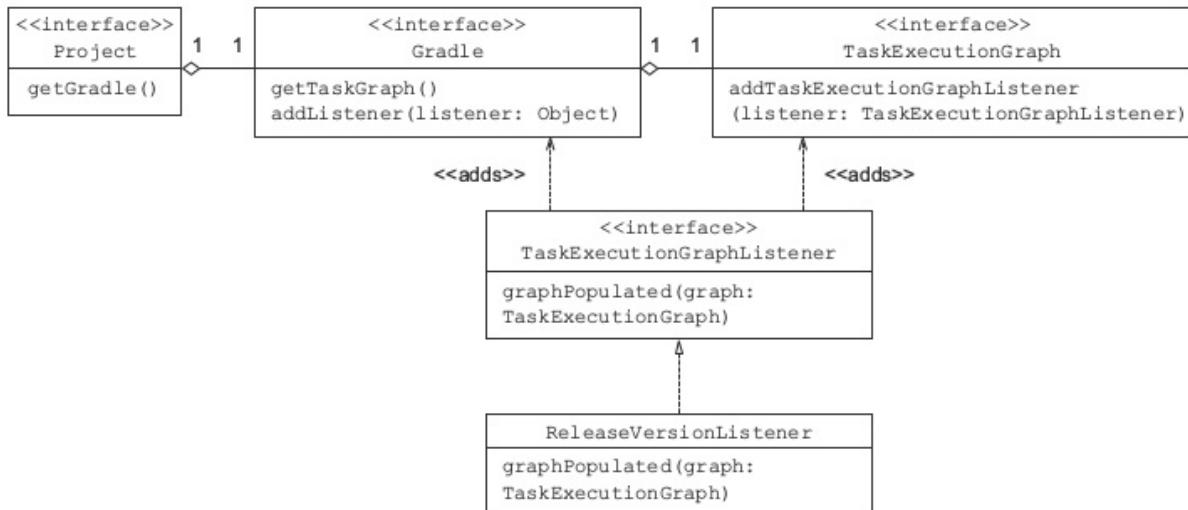
```
gradle.taskGraph.whenReady { TaskExecutionGraph taskGraph ->
    //检查任务图是否包括release任务
    if(taskGraph.hasTask(release)) {

        if(!version.release) {

            version.release = true

            ant.propertyfile(file: versionFile) {
                entry(key: 'release', type: 'string', operation: '=',
                    value: 'true')
            }
        }
    }
}
```

你也可以实现一个监听器来实现同样的效果，首先在构建脚本中编写一个实现指定监听器的类，然后在构建中注册这个实现，监听任务执行图的接口是TaskExecutionGraphListener，编写的时候你只需要实现graphPopulate(TaskExecutionGraph)方法，下图表示了这个过程：



下面是编程实现：

```
class ReleaseVersionListener implements TaskExecutionGraphListener {
    final static String releaseTaskPath = ':release'

    @Override

    void graphPopulated(TaskExecutionGraph taskGraph) {
        //查看是否包含release任务
        if(taskGraph.hasTask(releaseTaskPath)) {
            List<Task> allTasks = taskGraph.allTasks
            //查找release任务
            Task releaseTask = allTasks.find {it.path == releaseTaskPath }
            Project project = releaseTask.project

            if(!project.version.release) {

                project.version.release = true
                project.ant.propertyfile(file: project.versionFile) {
                    entry(key: 'release', type: 'string', operation: '=',
                           value: 'true')
                }
            }
        }
    }

    def releaseVersionListener = new ReleaseVersionListener()
    //注册监听器
    gradle.taskGraph.addTaskExecutionGraphListener(releaseVersionListener)
```

简介

在第三章我们在构建To Do应用的时候学习到了怎么声明对Servlet Api的依赖，Gradle的领域特定语言使得声明依赖和仓库变得很简单，你只需要在dependencies脚本中声明你所依赖的库，然后你需要告诉构建系统要从哪个仓库里下载依赖。提供了这两个信息，Gradle就能自动解析、下载依赖到你的电脑上，如果有需要会存储在本地缓存中必备以后需要。

这一章我们将介绍Gradle对依赖管理的强大支持，学习依赖分组和定位不同类型仓库的DSL元素。依赖管理看起来很容易，但是当出现依赖解析冲突时就会很棘手，复杂的依赖关系可能导致构建中依赖一个库的多个版本。Gradle通过分析依赖树得到依赖报告，你将很容易找到一个指定的依赖的来源，为什么选择这个版本来处理版本冲突。

Gradle有自己的依赖管理实现，为了避免其他依赖管理软件比如Ivy和Maven的缺点，Gradle关心的是性能、可靠性和复用性。

简要概述依赖管理

几乎所有基于JVM的项目都会或多或少依赖其他库，假设你在开发一个基于web的项目，你很可能会依赖很受欢迎的开源框架比如Spring MVC来提高效率。Java的第三方库一般以JAR文件的形式存在，一般用库名加版本号来标识。随着开发的进行依赖的第三方库增多小的项目变的越来越大，组织和管理你的JAR文件就很关键。

不算完美的依赖管理技术

由于Java语言并没提供依赖管理的工具，所以你的团队需要自己开发一套存储和检索依赖的想法。你可能会采取以下几种常见的方法：

- 手动复制JAR文件到目标机器，这是最原始的很容易出错的方法。
- 使用一个共享的存储介质来存储JAR文件(比如共享的网盘)，你可以加载网络硬盘或者通过FTP检索二进制文件。这种方法需要开发者事先建立好与仓库的连接，手动添加新的依赖到仓库中。
- 把依赖的JAR文件同源代码都添加到版本控制系统中。这种方法不需要任何额外的步骤，你的同伴在拷贝仓库的时候就能检索依赖的改变。另一方面，这些JAR文件占用了不必要的空间，当你的项目存在相互之间依赖的时候你需要频繁的check-in的检查源代码是否发生了改变。

自动管理依赖的重要性

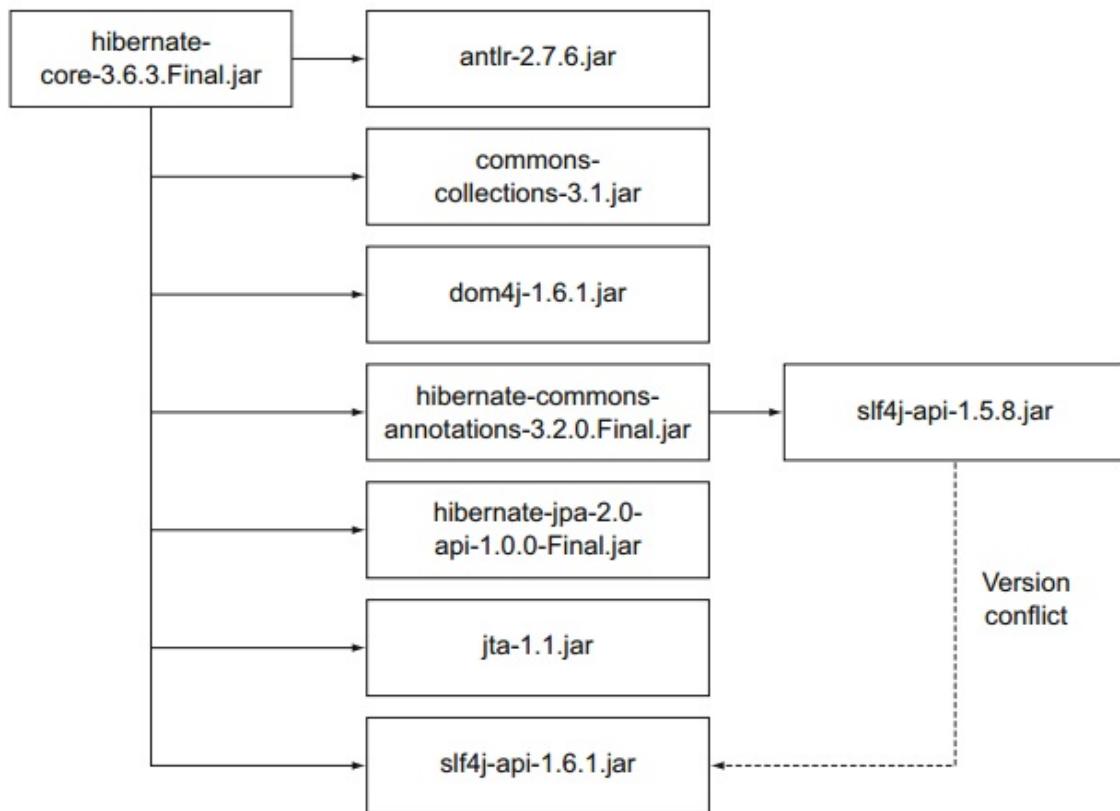
尽管上面的方法都能用，但是这距离理想的解决方案差远了，因为他们没有提供一个标准化的方法来命名和管理JAR文件。至少你得需要开发库的准确版本和它依赖的库(传递依赖)，这个为什么这么重要？

准确知道依赖的版本

如果在项目中你没有准确声明依赖的版本这将会是一个噩梦，如果没有文档你根本无法知道这个库支持哪些特性，是否升级一个库到新的版本就变成了一个猜谜游戏因为你不知道你的当前版本。

管理传递依赖

在项目的早期开发阶段传递依赖就会是一个隐患，这些库是第一层的依赖需要的，比如一个比较常见的开发方案是将Spring和Hibernate结合起来这会引入超过20个其他的开发库，一个库需要很多其他库来正常工作。下图展示了Hibernate核心库的依赖图：

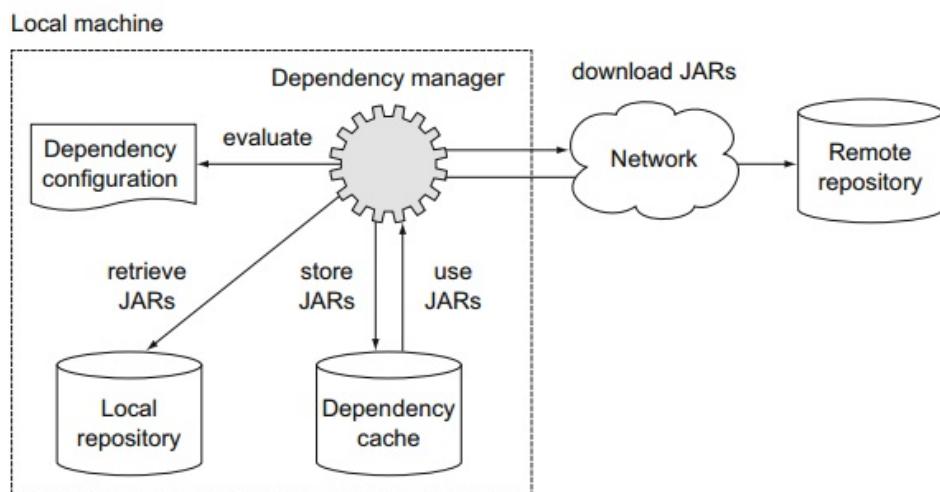


如果没有正确的管理依赖，你可以会遇到没想到过的编译期错误和运行期类加载问题。我们可以总结到我们需要一个更好的方式来管理依赖，一般来讲你想在项目元数据中声明你的依赖和它的版本号。作为一个项目自动化的过程，这个版本的库会自动从中央仓库下载、安装到你的项目中，我们来看几个现有的开源解决方案。

使用自动化的依赖管理

在Java领域里支持声明的自动依赖管理的有两个项目：Apache Ivy(Ant项目用的比较多的依赖管理器)和Maven(在构建框架中包含一个依赖管理器)，我不再详细介绍这两个的细节而是解释自动依赖管理的概念和机制。

Ivy和Maven是通过XML描述文件来表达依赖配置，配置包含两部分：依赖的标识加版本号和中央仓库的位置(可以是一个HTTP链接)，依赖管理器根据这个信息自动定位到需要下载的仓库然后下载到你的机器中。库可以定义传递依赖，依赖管理器足够聪明分析这个信息然后解析下载传递依赖。如果出现了依赖冲突比如上面的Hibernate core的例子，依赖管理器会试着解决。库一旦被下载就会存储在本地的缓存中，构建系统先检查本地缓存中是否存在需要的库然后再从远程仓库中下载。下图显示了依赖管理的关键元素：



Gradle通过DSL来描述依赖配置，实现了上面描述的架构。

自动依赖管理面临的挑战

虽然依赖管理器简化了手工的操作，但有时也会遇到问题。你会发现你的依赖图中会依赖同一个库的不同版本，使用日志框架经常会遇到这个问题，依赖管理器基于一个特定的解决方案只选择其中一个版本来避免版本冲突。如果你想知道某个库引入了什么版本的传递依赖，Gradle提供了一个非常有用的依赖报告来回答这个问题。下一节我会通过一个例子来讲解。

依赖管理实战

在前面我们学习了怎么使用Jetty插件来使用自带的Jetty容器来部署一个TODo应用，Jetty是一个轻量级的开发容器，启动非常快。很多企业级的应用都使用其他的Web容器来部署应用，假设你使用的是Apache Tomcat。

声明依赖

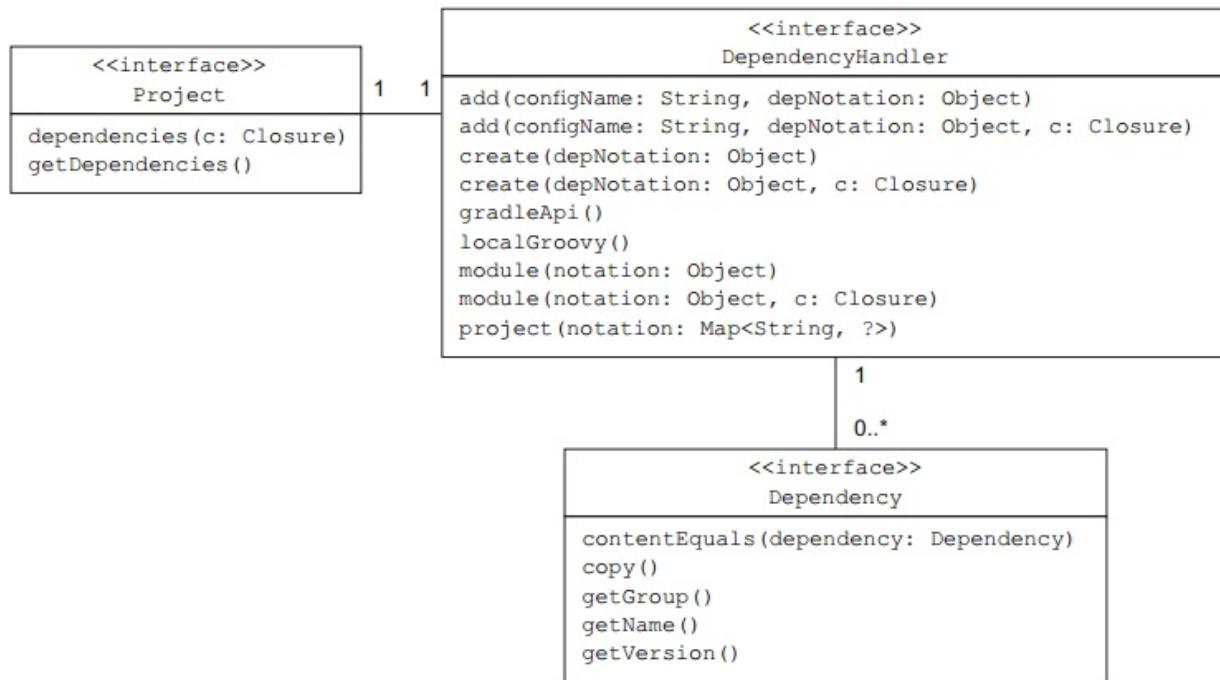
DSL配置block dependencies用来给配置添加一个或多个依赖，你的项目不仅可以添加外部依赖，下面这张表显示了Gradle支持的各种不同类型的依赖。

Type	Description
External module dependency	A dependency on an external library in a repository including its provided metadata
Project dependency	A dependency on another Gradle project
File dependency	A dependency on a set of files in the file system
Client module dependency	A dependency on an external library in a repository with the ability to declare the metadata yourself
Gradle runtime dependency	A dependency on Gradle's API or a library shipped with the Gradle runtime

这一章直接扫外部模块依赖和文件依赖，我们来看看Gradle API是怎么表示依赖的。

理解依赖的API表示

每个Gradle项目都有一个DependencyHandler的实例，你可以通过getDependencies()方法来获取依赖处理器的引用，上表中每一种依赖类型在依赖处理器中都有一个相对应的方法。每一个依赖都是Dependency的一个实例，group, name, version, 和classifier这几个属性用来标识一个依赖，下图清晰的表示了项目(Project)、依赖处理器(DependencyHandler)和依赖三者之间的关系：



外部模块依赖

在Gradle的术语里，外部库通常是以JAR文件的形式存在，称之为外部模块依赖，代表项目层次外的一个模块，这种类型的依赖是通过属性来唯一的标识，接下来我们来介绍每个属性的作用。

依赖属性

当依赖管理器从仓库中查找依赖时，需要通过属性的结合来定位，最少需要提供一个name。

- group：这个属性用来标识一个组织、公司或者项目，可以用点号分隔，Hibernate的group是org.hibernate。
- name：name属性唯一的描述了这个依赖，hibernate的核心库名称是hibernate-core。
- version：一个库可以有很多个版本，通常会包含一个主版本号和次版本号，比如Hibernate核心库3.6.3-Final。
- classifier：有时候需要另外一个属性来进一步的说明，比如说明运行时的环境，Hibernate核心库没有提供classifier。

依赖的写法

你可以使用下面的语法在项目中声明依赖：

```

dependencies {
    configurationName dependencyNotation1,      dependencyNotation2, ...
}
  
```

你先声明你要给哪个配置添加依赖，然后添加依赖列表，你可以用map的形式来注明，你也可以直接用冒号来分隔属性，比如这样的：

```
org.hibernate:hibernate-core:3.6.3-Final
  _____|_____|_____
  group      name     version
```

```
//声明外部属性
ext.cargoGroup = 'org.codehaus.cargo'
ext.cargoVersion = '1.3.1'

dependencies {
    //使用映射声明依赖
    compile group: cargoGroup, name: 'cargo-core-uberjar',version: cargoVersion
    //用快捷方式来声明，引用了前面定义的外部属性
    cargo "$cargoGroup:cargo-ant:$cargoVersion"
}
```

如果你项目中依赖比较多，你把一些共同的依赖属性定义成外部属性可以简化build脚本。

Gradle没有给项目选择默认的仓库，当你没有配置仓库的时候运行deployToLocalTomcat任务的时候回出现如下的错误：

```
$ gradle deployToLocalTomcat
:deployToLocalTomcat FAILED
FAILURE: Build failed with an exception.

Where: Build file '/Users/benjamin/gradle-in-action/code/chapter5/cargo-configuration/bui
What went wrong:
Execution failed for task ':deployToLocalTomcat'.
> Could not resolve all dependencies for configuration ':cargo'.
    > Could not find group:org.codehaus.cargo, module:cargo-core-uberjar, version:1.3.1.
      Required by:
          :cargo-configuration:unspecified
> Could not find group:org.codehaus.cargo, module:cargo-ant,version:1.3.1.
      Required by:
          :cargo-configuration:unspecified
```

到目前为止还没讲到怎么配置不同类型的仓库，比如你想使用MavenCentral仓库，添加下面的配置代码到你的build脚本中：

```
repositories {
    mavenCentral()
}
```

检查依赖报告

当你运行`dependencies`任务时，这个依赖树会打印出来，依赖树显示了你`build`脚本声明的顶级依赖和它们的传递依赖：

```
$ gradle dependencies
:dependencies

-----
Root project
-----

cargo - Classpath for Cargo Ant tasks.
+--- org.codehaus.cargo:cargo-core-uberjar:1.3.1
|   +--- commons-discovery:commons-discovery:0.4
|   |   \--- commons-logging:commons-logging:1.0.4
|   +--- jdom:jdom:1.0
|   +--- dom4j:dom4j:1.4
|   |   +--- xml-apis:xml-apis:1.0.b2 -> 1.3.03
|   |   +--- jaxen:jaxen:1.0-FCS
|   |   +--- saxpath:sxpath:1.0-FCS
|   |   +--- msv:msv:20020414
|   |   +--- relaxngDatatype:relaxngDatatype:20020414
|
|   |   \--- isorelax:isorelax:20020414
+--- jaxen:jaxen:1.0-FCS (*)
+--- saxpath:sxpath:1.0-FCS (*)
+--- msv:msv:20020414 (*)
+--- relaxngDatatype:relaxngDatatype:20020414 (*)
+--- isorelax:isorelax:20020414 (*)
+--- com.sun.xml.bind:jaxb-impl:2.1.13
|   +--- javax.xml.bind:jaxb-api:2.1
|   |   +--- javax.xml.stream:stax-api:1.0-2
|   |   \--- javax.activation:activation:1.1
+--- javax.xml.bind:jaxb-api:2.1 (*)
+--- javax.xml.stream:stax-api:1.0-2 (*)
+--- javax.activation:activation:1.1 (*)
+--- org.apache.ant:ant:1.7.1
|   \--- org.apache.ant:ant-launcher:1.7.1
+--- org.apache.ant:ant-launcher:1.7.1 (*)
+--- xerces:xercesImpl:2.8.1
|   \--- xml-apis:xml-apis:1.3.03 (*)
+--- xml-apis:xml-apis:1.3.03 (*)
\--- commons-logging:commons-logging:1.0.4 (*)
\--- org.codehaus.cargo:cargo-ant:1.3.1
    \--- org.codehaus.cargo:cargo-core-uberjar:1.3.1 (*)
```

仔细观察你会发现有些传递依赖标注了`*`号，表示这个依赖被忽略了，这是因为其他顶级依赖中也依赖了这个传递的依赖，Gradle会自动分析下载最合适的依赖。

排除传递依赖

Gradle允许你完全控制传递依赖，你可以选择排除全部的传递依赖也可以排除指定的依赖，假设你不想使用UberJar传递的xml-api的版本而想声明一个不同版本，你可以使用exclude方法来排除它：

```
dependencies {
    cargo('org.codehaus.cargo:cargo-ant:1.3.1') {
        exclude group: 'xml-apis', module: 'xml-apis'
    }
    cargo 'xml-apis:xml-apis:2.0.2'
}
```

exclude属性值和正常的依赖声明不太一样，你只需要声明group和(或)module，Gradle不允许你只排除指定版本的依赖。

有时候仓库中找不到项目依赖的传递依赖，这会导致构建失败，Gradle允许你使用transitive属性来排除所有的传递依赖：

```
dependencies {
    cargo('org.codehaus.cargo:cargo-ant:1.3.1') {
        transitive = false
    }
    // 选择性的声明一些需要的库
}
```

动态版本声明

如果你想使用一个依赖的最新版本，你可以使用latest.integration，比如声明 Cargo Ant tasks 的最新版本，你可以这样写 `org.codehaus.cargo:cargo-ant:latest-integration`，你也可以用一个+号来动态的声明：

```
dependencies {
    //依赖最新的1.x版本
    cargo 'org.codehaus.cargo:cargo-ant:1.+'
```

Gradle的dependencies任务可以清晰的看到选择了哪个版本，这里选择了1.3.1版本：

```
$ gradle -q dependencies
-----
Root project
-----
Listing 5.4 Excluding a single dependency
Listing 5.5 Excluding all transitive dependencies
Listing 5.6 Declaring a dependency on the latest Cargo 1.x version
Exclusions can be
declared in a shortcut
or map notation.
120 CHAPTER 5 Dependency management
cargo - Classpath for Cargo Ant tasks.
\--- org.codehaus.cargo:cargo-ant:1.+ -> 1.3.1
\--- ...
```

文件依赖

如果你没有使用自动的依赖管理工具，你可能会把外部库作为源代码的一部分或者保存在本地文件系统中，当你想把项目迁移到Gradle的时候，你不想去重构，Gradle很简单就能配置文件依赖。下面这段代码复制从Maven中央仓库解析的依赖到libs/cargo目录。

```
task copyDependenciesToLocalDir(type: Copy) {
    //Gradle提供的语法糖
    from configurations.cargo.asFileTree
    into "${System.properties['user.home']}/libs/cargo"
}
```

运行这个任务之后你就可以在依赖中声明Cargo库了，下面这段代码展示了怎么给cargo配置添加JAR文件依赖：

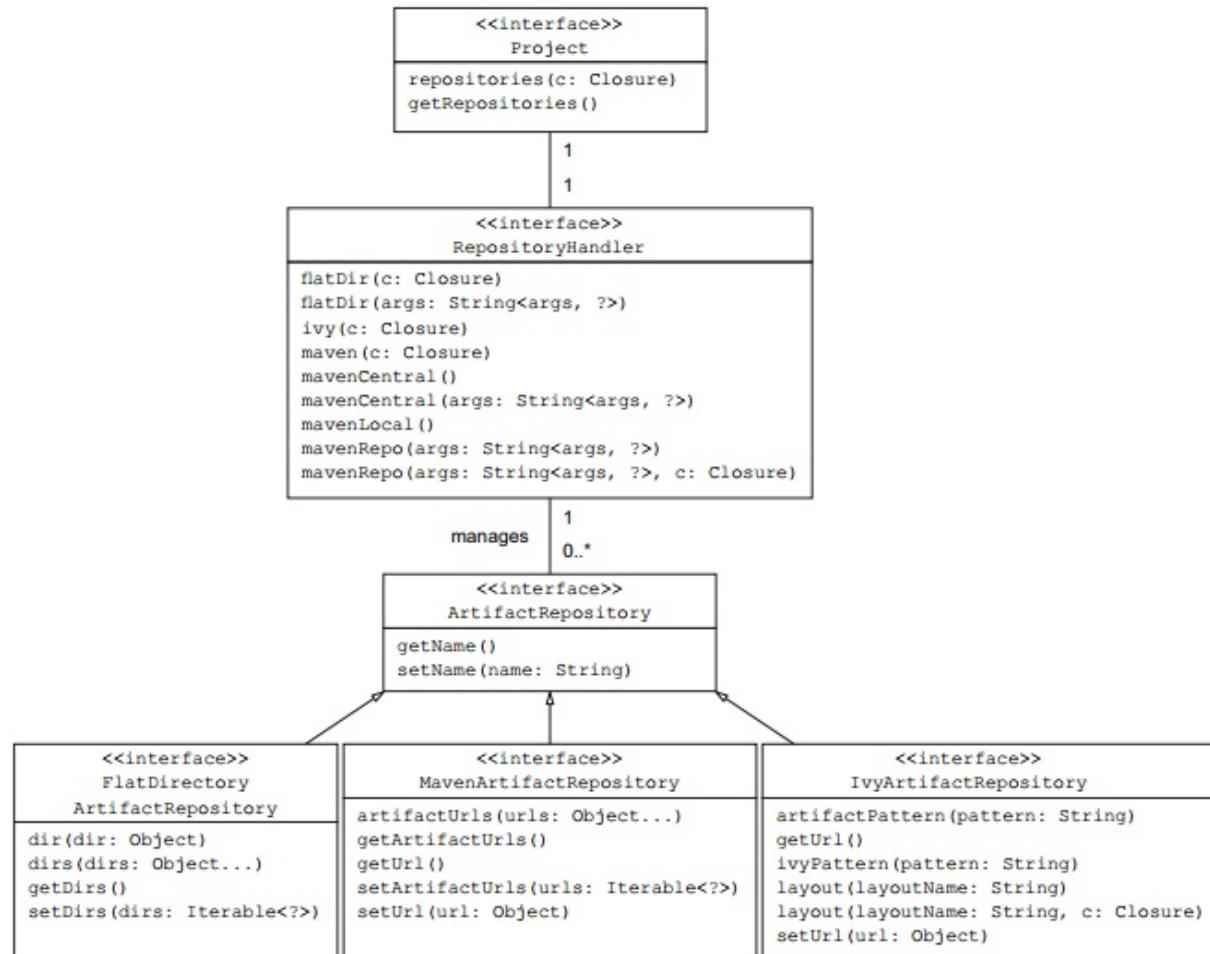
```
dependencies {
    cargo fileTree(dir: "${System.properties['user.home']}/libs/cargo", include: '*.jar')
}
```

配置远程仓库

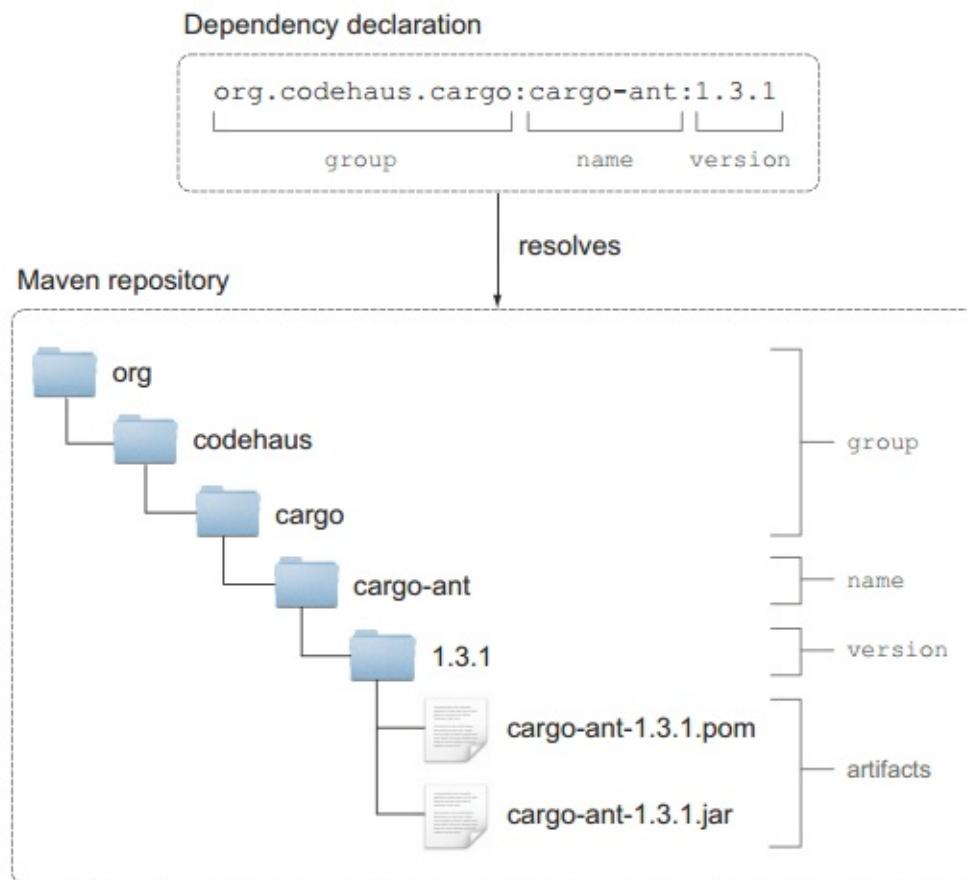
Gradle支持下面三种不同类型的仓库：

Type	Description
Maven repository	A Maven repository on the local file system or a remote server, or the preconfigured Maven Central
Ivy repository	An Ivy repository on the local file system or a remote server with a specific layout pattern
Flat directory repository	A repository on the local file system without metadata support

下图是配置不同仓库对应的Gradle API：



下面以Maven仓库来介绍， Maven仓库是Java项目中使用最为广泛的一个仓库， 库文件一般是以JAR文件的形式存在， 用XML(POM文件)来描述库的元数据和它的传递依赖。所有的库文件都存储在仓库的指定位置， 当你在构建脚本中声明了依赖时， 这些属性用来找到库文件在仓库中的准确位置。group属性标识了Maven仓库中的一个子目录， 下图展示了Cargo依赖属性是怎么对应到仓库中的文件的：



RepositoryHandler接口提供了两个方法来定义Maven仓库， mavenCentral方法添加一个指向仓库列表的引用， mavenLocal方法引用你文件系统中的本地Maven仓库。

添加Maven仓库

要使用Maven仓库你只需要调用mavenCentral方法， 如下所示：

```
repositories {
    mavenCentral()
}
```

添加本地仓库

本地仓库默认在`/.m2/repository`目录下， 只需要添加如下脚本来引用它：

```
repositories {  
    mavenLocal()  
}
```

添加自定义**Maven**仓库

如果指定的依赖不存在与**Maven**仓库或者你想通过建立自己的企业仓库来确保可靠性，你可以使用自定义的仓库。仓库管理器允许你使用**Maven**布局来配置一个仓库，这意味着你要遵守**artifact**的存储模式。你也可以添加验证凭证来提供访问权限，**Gradle**的API提供两种方法配置自定义的仓库：**maven()**和**mavenRepo()**。下面这段代码添加了一个自定义的仓库，如果**Maven**仓库中不存在相应的库会从自定义仓库中查找：

```
repositories {  
    mavenCentral()  
    maven {  
        name 'Custom Maven Repository'  
        url 'http://repository.forge.cloudbees.com/release/'  
    }  
}
```

简介

每一个活跃的项目会随着时间慢慢增长的，一开始可能只是个很小的项目到后面可能包含很多包和类。为了提高可维护性和解耦的目的，你可能想把项目根据逻辑和功能来划分成一个个模块。模块通常按照等级来组织，相互之间可以定义依赖。

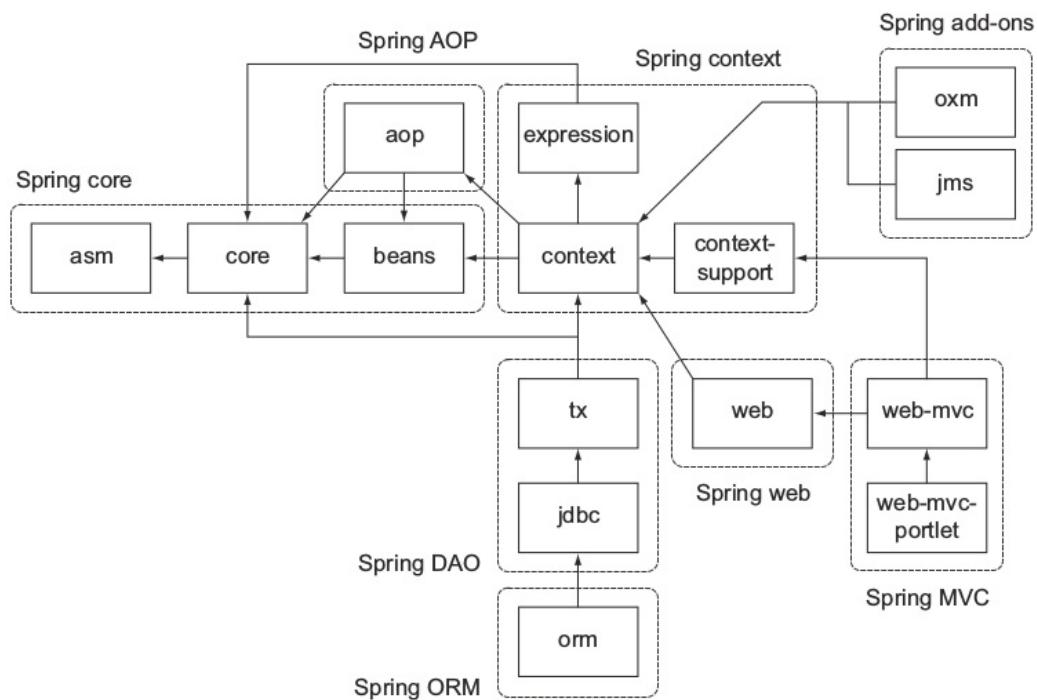
Gradle给项目模块化提供了强大的支持，在Gradle中每个模块都是一个项目，我们称之为多项目构建，这一章介绍Gradle的多项目构建。

项目模块化

在企业项目中，包层次和类关系比较负责，把代码拆分成模块是一个比较困难的任务，因为这需要你清晰的划分功能的边界，比如把业务逻辑和数据持久化拆分开来。

解耦和聚合

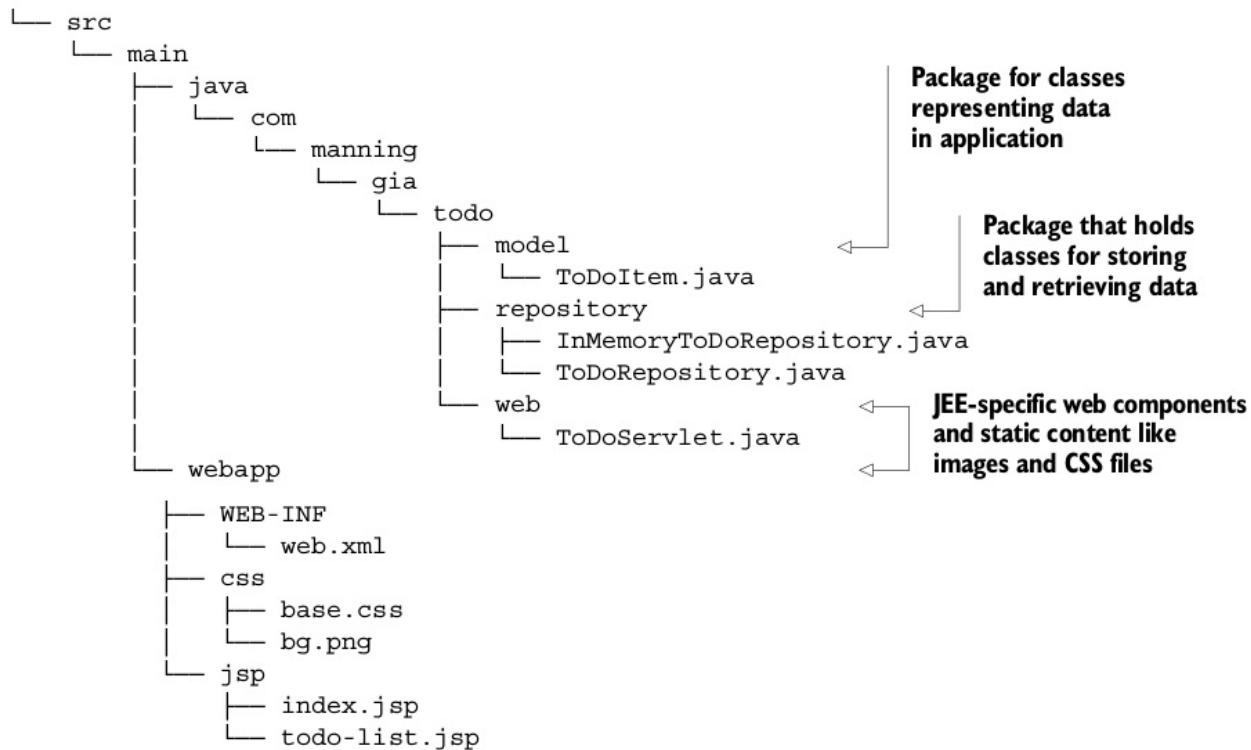
但你的项目符合高内聚低耦合时，模块化就变得很容易，这是一条非常好的软件开发实践。一个很好的模块化的例子就是Spring框架，spring框架提供了很多服务，比如MVC web框架、事务管理器、JDBC数据库连接等，下图展示了Spring3.x的模块间的关系：



看起来这个架构非常的庞大吓人，定义了非常多的组件相互间的依赖关系比较复杂，在实际使用过程中你并不需要导入整个框架到你的项目中，你可以选择你需要使用的服务。幸运的是模块之间的依赖都是通过元数据指定的，Gradle的依赖管理很容易解析它的传递依赖。

区分模块

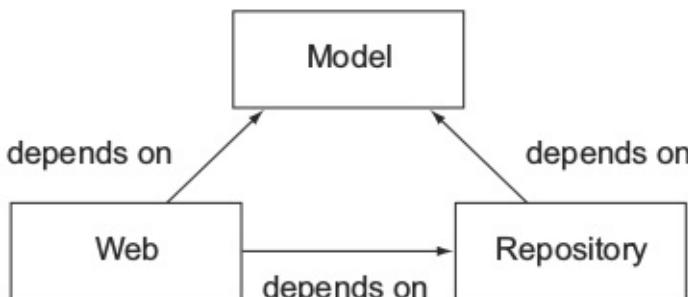
下面我们还是继续之前那个ToDo的例子，我们来把它拆分为多个模块。



你已经根据类的功能把它们拆分成一个个包，基本上分为下面几个功能：

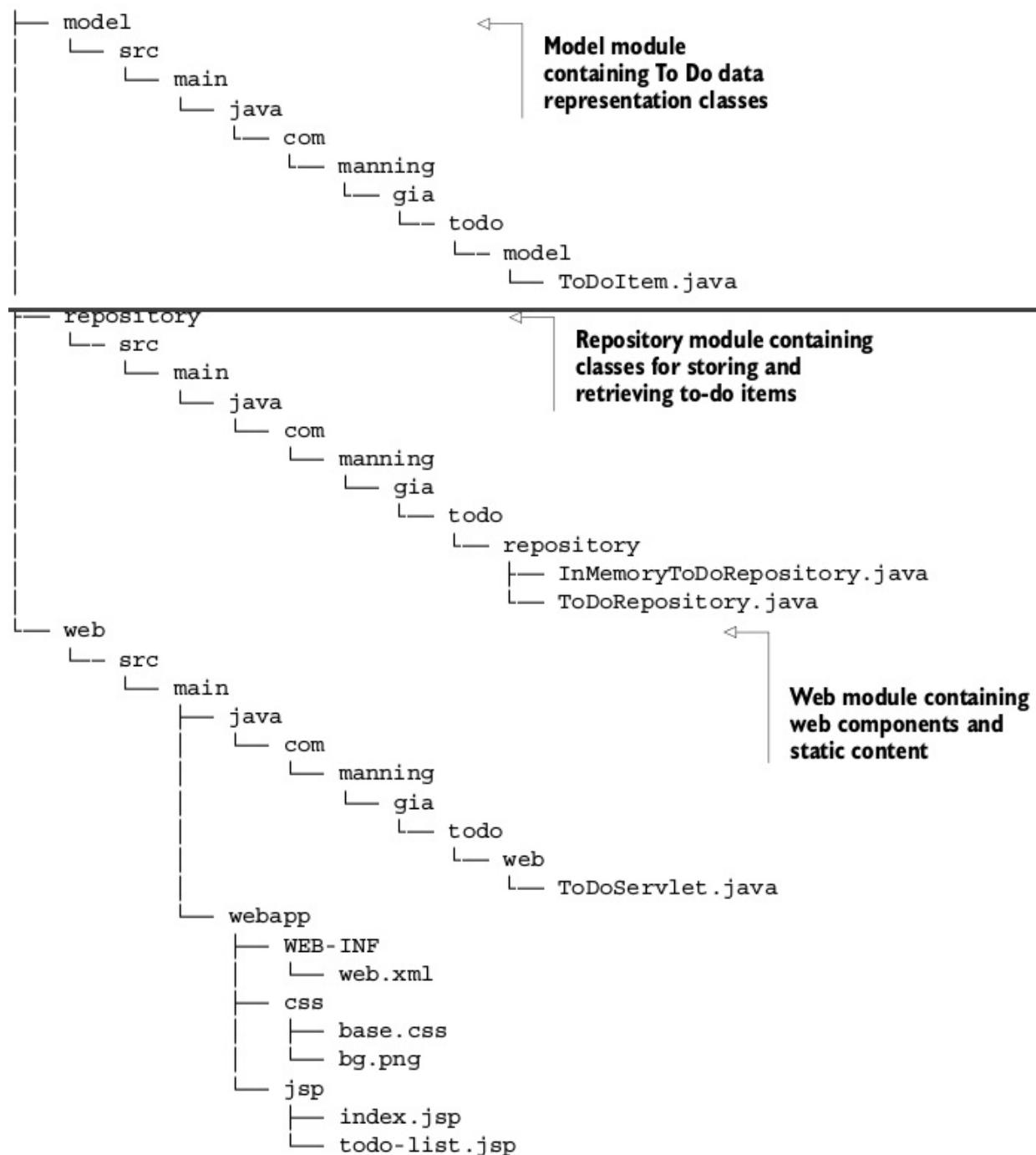
- 模型: 用来表示数据
- 仓库: 检索和存储数据
- Web: 用来处理HTTP请求、渲染页面的Web组件

虽然这是一个非常小的项目，这些模块之间也有依赖关系：



重构模块

现在很容易把存在的项目结构重构为几个模块，对于每个模块，使用合适的名称创建一个子目录，把相关的代码移动到里面。默认的源代码路径src/main/java还是毫发无损，Web模块需要默认的web应用源代码目录src/main/webapp,下面显示了模块化的项目布局：



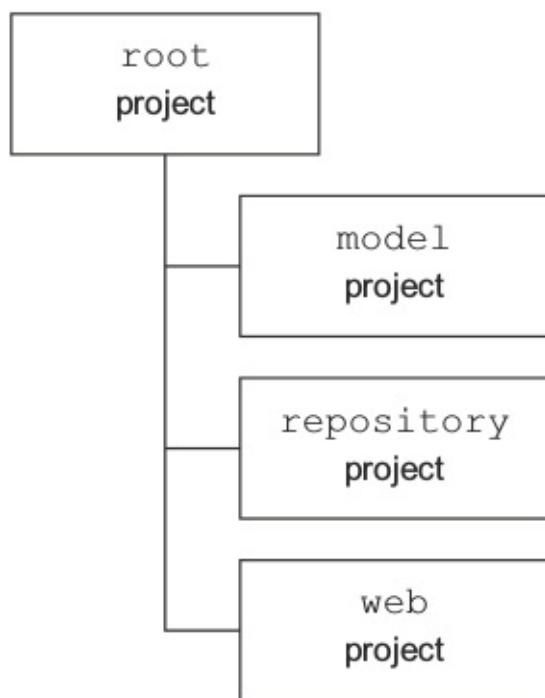
多项目打包

上一节你给你的项目定义了一个层次化的目录结构，整个项目包含一个根目录和每个模块一个子目录，这一节你将学习怎么用Gradle来构建这样一个项目结构。

首先在你的根目录新建一个build.gradle文件，创建一个空的build脚本然后运行gradle projects:

```
$ gradle projects
:projects
-----
Root project
-----
Root project 'todo'
No sub-projects
```

接下来学习怎么通过settings.gradle来定义项目的子项目。



介绍设置文件

设置文件用来表示项目的层次结构，默认的设置文件名称是settings.gradle，对于你想添加的每一个子项目，调用include方法来添加。 //参数是项目路径，不是文件路径 include 'model'
include 'repository', 'web'

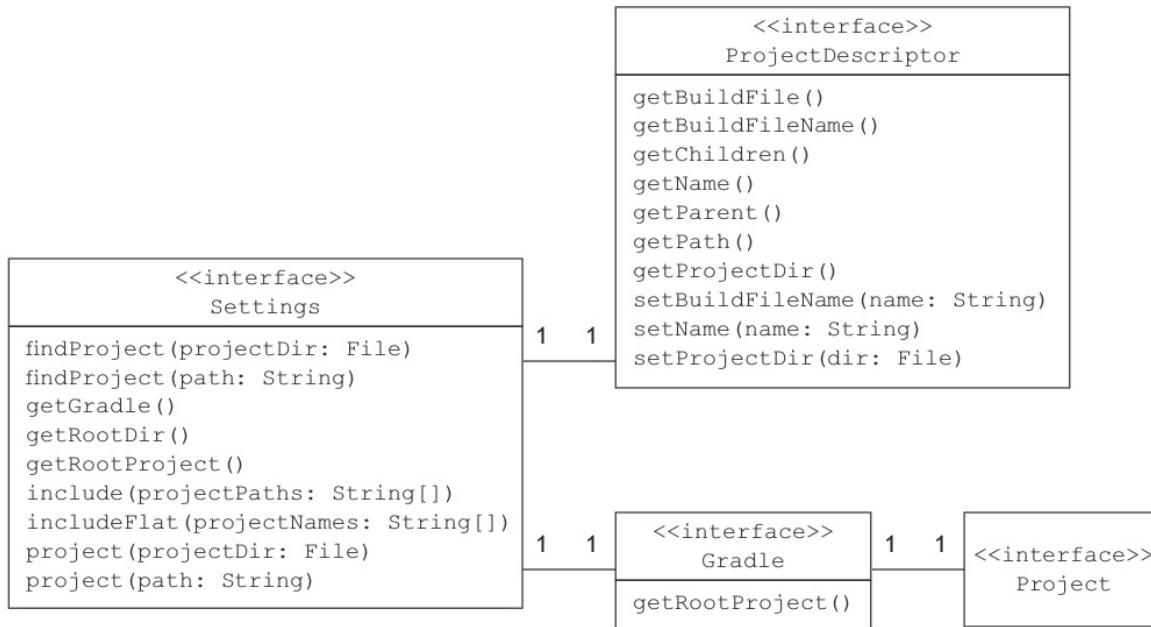
提供的项目路径是相对于根目录，记住冒号：是用来分隔目录层次，比如你想表示 model/todo/items 这个目录，在 gradle 里面是 model:todo:items 这样表示。接下来执行 gradle projects 任务：

```
$ gradle projects
:projects
-----
Root project
-----
Root project 'todo'
+-- Project ':model'
+-- Project ':repository'
+-- Project ':web'
```

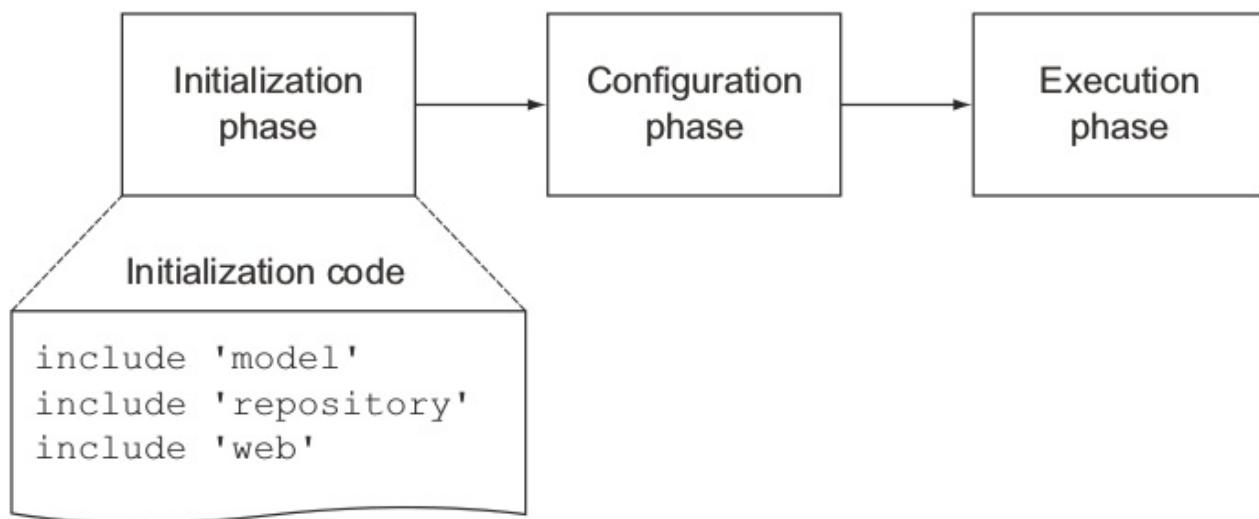
通过添加这个设置文件，你就创建了一个多项目的构建包含一个根项目和三个子项目，不需要额外的配置。

理解 settings 的 API 表示

在 Gradle 开始执行构建之前，它创建一个 Settings 类型的实例，Settings 接口直接用来表示 settings 文件，主要目的是通过代码来动态添加项目参与到多项目构建中，下图是你可以访问的 Gradle API。



之前我们介绍过 Gradle 有三个生命周期，实例化阶段->配置阶段->执行阶段，Settings 处于实例化阶段，Gradle 自动找出一个子项目是否处在一个多项目构建中。

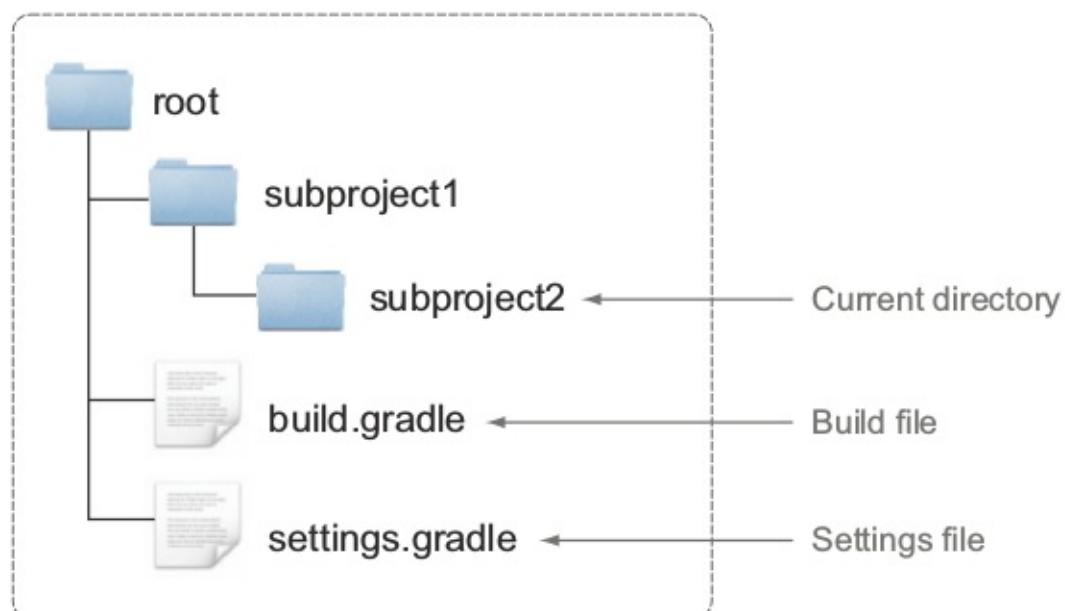
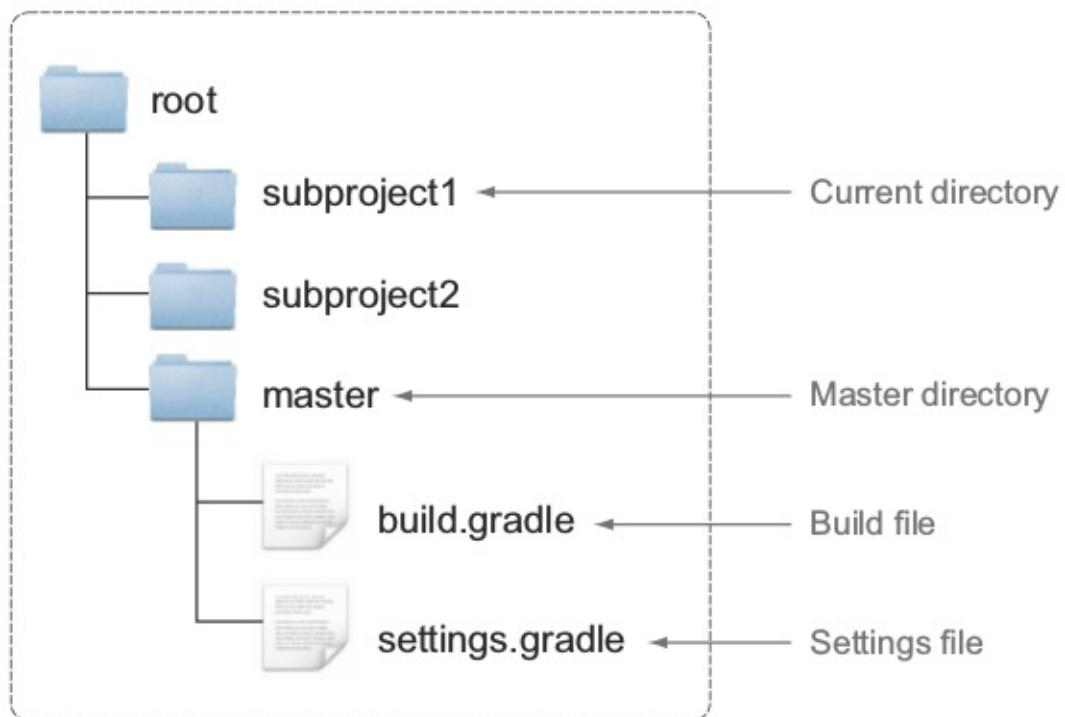


设置文件解析

Gradle允许你从根目录或者任何子目录中运行构建任务，只要它包含一个build脚本，Gradle怎么知道一个子项目是不是一个多层次构建的一部分呢？他需要查找settings文件，Gradle通过两步来查找设置文件。

1. Gradle查找和当前目录具有相同嵌套级别的master目录下的设置文件
2. 如果第一步没有找到设置文件，Gradle从当前目录开始逐步查找父目录

如果找到了settings文件，项目包含在另一个项目中，这个项目就当成是多层次构建的一部分。整个过程如下所示：



配置子项目

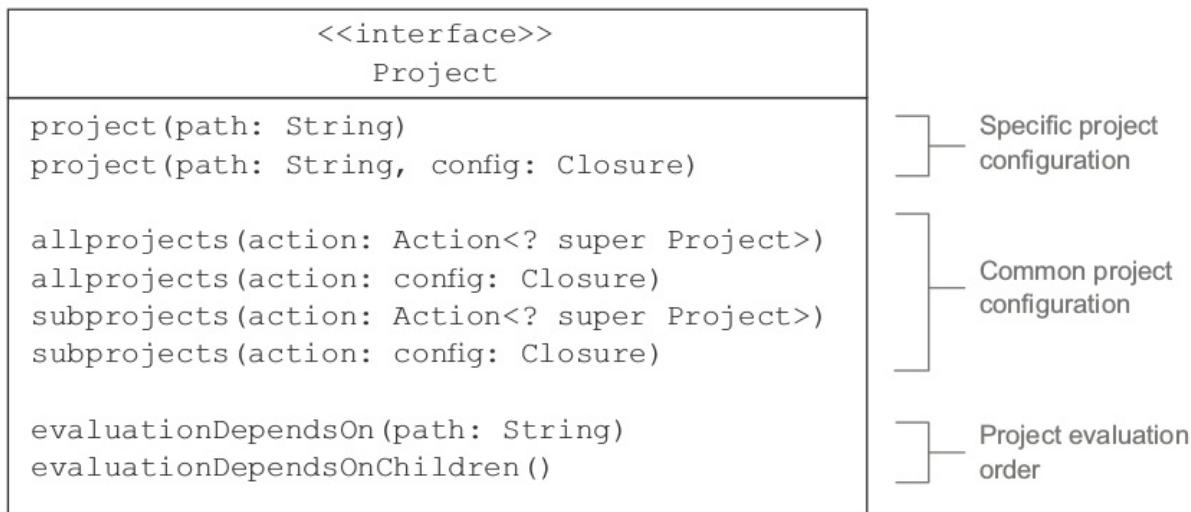
到目前为止你已经把ToDo项目根据功能拆分成多个模块，接下来可以用之前的方法来定义构建逻辑，下面有几点需要主要：

- 根目录和子目录使用相同的group和version属性值
- 所有的子目录都是Java项目需要Java插件来正常工作，所以你只需要在子项目中应用Java插件
- web子项目是唯一一个依赖外部库的项目，它需要打包成WAR而不是JAR
- 子项目之间可以定义模块依赖

接下来你将学习如何定义特定的和共有的构建逻辑，怎么样去避免重复的配置。有些子项目可能依赖其他项目的源代码，比如repository项目依赖model项目，通过声明项目依赖可以避免拷贝源代码。

理解项目的API表示

之前我介绍过项目Project可能会用到的一些API，接下来还有一些API用在多项目构建中。`project`方法用于声明指定项目的构建代码，需要提供项目的路径，比如`:model`。有时候你想给所有的项目或者只有子项目定义一些逻辑，你可以使用`allprojects`和`subprojects`方法，比如你想给所有的子项目添加Java插件，你需要使用`subprojects`方法。



定义项目特有的行为

指定项目的行为通过`project`方法来定义，为了给三个子项目model、repository、web定义构建逻辑，你需要给它们分别创建一个配置块。下面`build.gradle`文件：

```
ext.projectIds = ['group': 'com.manning.gia', 'version': '0.1']

group = projectIds.group
version = projectIds.version

project(':model') {
    group = projectIds.group
    version = projectIds.version
    apply plugin: 'java'
}

project(':repository') {
    group = projectIds.group
    version = projectIds.version
    apply plugin: 'java'
}

project(':web') {
    group = projectIds.group
    version = projectIds.version
    apply plugin: 'java'
    apply plugin: 'war'
    apply plugin: 'jetty'

repositories {
    mavenCentral()
}

dependencies {
    providedCompile 'javax.servlet:servlet-api:2.5'
    runtime 'javax.servlet:jstl:1.1.2'
}
}
```

从整个项目的根目录那里，你可以执行子项目的任务，需要记住项目路径是通过冒号分隔，比如你想执行model子项目的build任务，在命令行中执行gradle :model:build就可以，如下所示：

```
$ gradle :model:build
:model:compileJava
:model:processResources UP-TO-DATE
:model:classes
:model:jar
:model:assemble
:model:compileTestJava UP-TO-DATE
:model:processTestResources UP-TO-DATE
:model:testClasses UP-TO-DATE
:model:test
:model:check
:model:build
```

声明项目依赖

声明项目依赖和声明外部依赖非常类似，只需要在dependencies配置块中声明，如下所示：

```
project(':model') {
...
}

project(':repository') {
...

dependencies {
    //声明编译期依赖项目model
    compile project(':model')

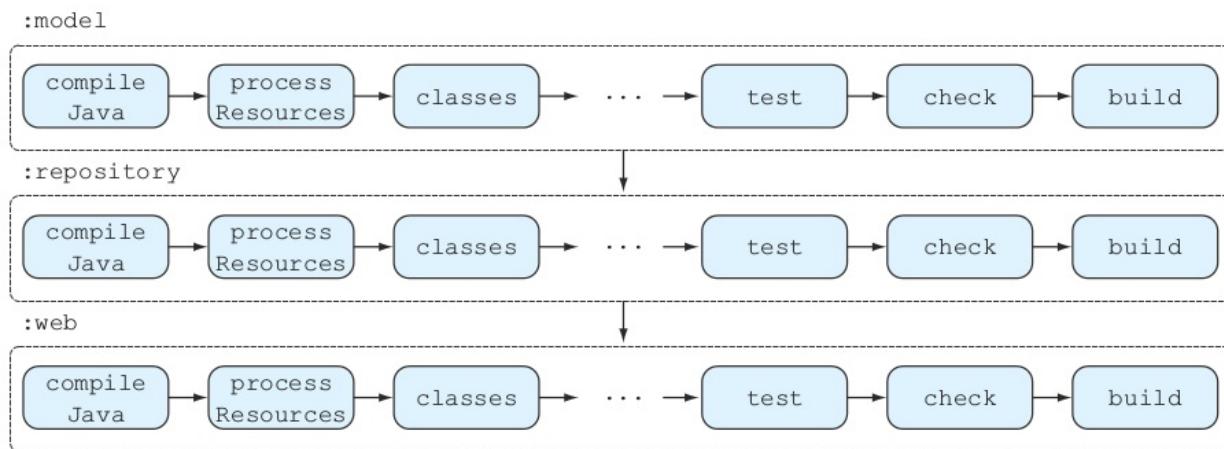
}
}

project(':web') {
...

dependencies {
    //声明编译期依赖项目repository
    compile project(':repository')
    providedCompile 'javax.servlet:servlet-api:2.5'
    runtime 'javax.servlet:jstl:1.1.2'
}
}
```

这样就定义了我们的项目依赖，注意当一个项目依赖于另一个项目时，另一个项目的项目依赖和外部依赖同样会被添加进来，在构建周期的初始化阶段，Gradle决定项目的执行顺序。

从根目录的执行顺序是这样的：

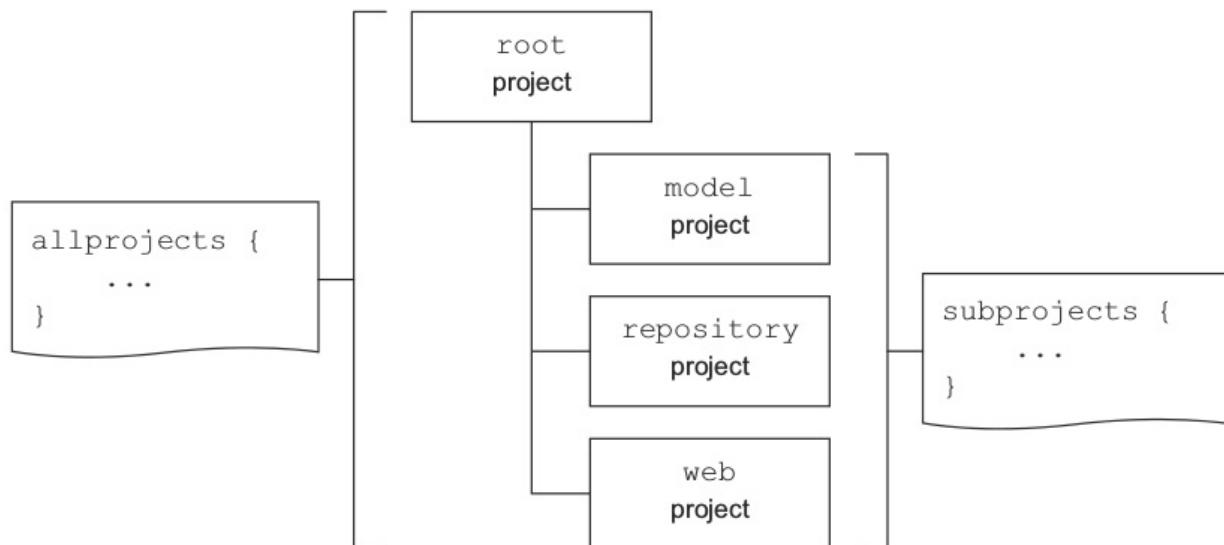


有时候你并不需要重新构建那些并未改变的项目，Gradle提供了部分构建partial builds的选项，通过命令行选项-a或者--no-rebuild。比如你只改变了repository项目不想重新构建model子项目，你可以这样做：gradle :repository:build -a,如下所示：

```
$ gradle :repository:build -a
:repository:compileJava
:repository:processResources UP-TO-DATE
:repository:classes
:repository:jar
:repository:assemble
:repository:compileTestJava UP-TO-DATE
:repository:processTestResources UP-TO-DATE
:repository:testClasses UP-TO-DATE
:repository:test
:repository:check
:repository:build
```

定义共同的行为

上面你在所有的子项目中添加了Java插件，给所有的项目添加了一个外部属性ProjectIds,当你的子项目变得比较多的时候这样子做可能是个问题，接下来你可以通过allprojects和subprojects方法来改善你的构建代码。



这是什么意思？这意味着你可以用`allprojects`方法给所有的项目添加`group`和`version`属性，由于根项目不需要Java插件，你可以使用`subprojects`给所有子项目添加Java插件，如下所示：

```

allprojects {
    group = 'com.manning.gia'
    version = '0.1'
}

subprojects {
    apply plugin: 'java'
}

project(':repository') {
    dependencies {
        compile project(':model')
    }
}

project(':web') {
    apply plugin: 'war'
    apply plugin: 'jetty'

    repositories {
        mavenCentral()
    }

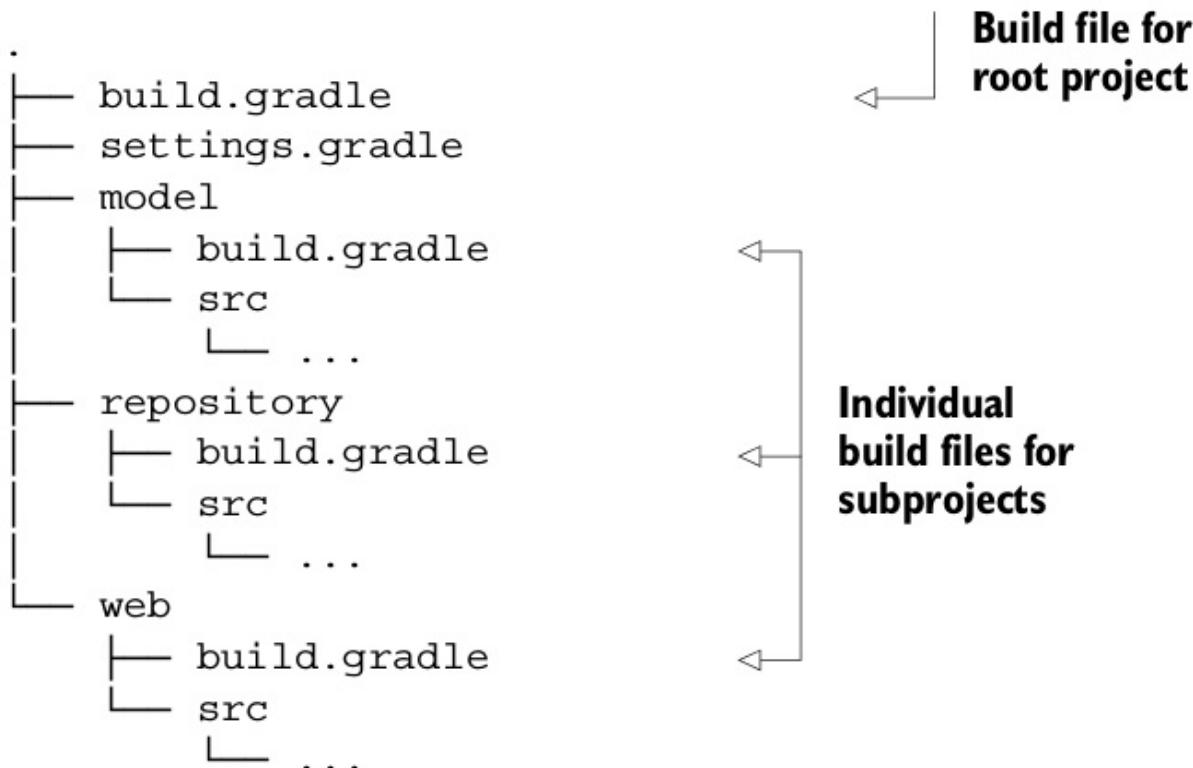
    dependencies {
        compile project(':repository')
        providedCompile 'javax.servlet:servlet-api:2.5'
        runtime 'javax.servlet:jstl:1.1.2'
    }
}

```

拆分项目文件

到目前为止我们自定义了一个build.gradle和settings.gradle文件，随着你添加越来越多的子项目和任务到build.gradle中，代码的维护性将会下降。通过给每个子项目建立一个单独的build.gradle文件可以解决这个问题。

接下来我们在每个子项目的目录下创建一个build.gradle文件，目录如下：



现在你可以把构建逻辑从原先的build脚本中拆分开来放到合适的位置。

定义根项目的构建代码

移除了那些与特定子项目相关的代码后，根项目的内容变得非常简单，只需要保留`allprojects`和`subprojects`配置块，如下所示：

```
allprojects {  
    group = 'com.manning.gia'  
    version = '0.1'  
}  
  
subprojects {  
    apply plugin: 'java'  
}
```

定义子项目的构建代码

接下来你只需要把与特定项目相关的构建代码移到相应的build.gradle文件里就可以了，如下所示：

repository子项目的构建代码：

```
dependencies {  
    compile project(':model')  
}
```

web子项目的构建代码：

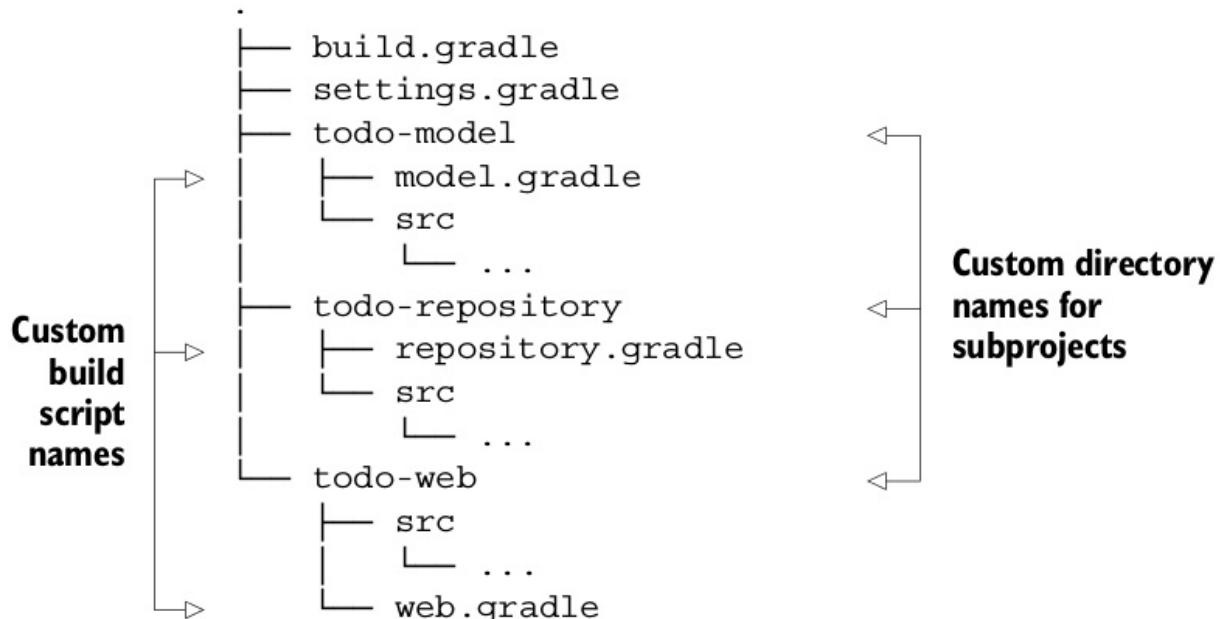
```
apply plugin: 'war'  
apply plugin: 'jetty'  
  
repositories {  
    mavenCentral()  
}  
  
dependencies {  
    compile project(':repository')  
    providedCompile 'javax.servlet:servlet-api:2.5'  
    runtime 'javax.servlet:jstl:1.1.2'  
}
```

运行这个多项目构建和之前单独的一个build脚本的结果完全一样，当时你该上了构建代码的可读性和维护性。

自定义脚本

Gradle构建脚本的标准名称是build.gradle，在一个子项目构建的环境中，你想自定义你的构建脚本名称来显得高大上一点，因为多个项目有相同的构建脚本名称可能会混淆，接下来介绍如何使用自定义的脚本名称。

还是之前那个例子，假设所有的子项目路径都是以todo-开头，比如web子项目就是在todo-web目录下，构建脚本名称应该清晰的表示它的作用，如下图所示：



要使这个结构起作用关键点就是settings文件，它提供了除了包含哪个子目录的其他功能，实际上设置文件是一个构建脚本，它会在构建生命周期的评估阶段执行，通过Gradle提供的API来添加自定义的逻辑，如下所示：

```

//通过目录来添加子项目
include 'todo-model', 'todo-repository', 'todo-web'

//设置根项目的名字
rootProject.name = 'todo'

//迭代访问所有根目录下的子项目，设置自定义的构建脚本名称
rootProject.children.each {
    it.buildFileName = it.name + '.gradle' - 'todo-'
}
  
```

总结

通过把系统中的项目模块化可以提高代码的复用性、维护性，符合高内聚低耦合的软件开发准则。

简介

在之前的章节我们实现了一个简单但是功能齐全的web项目、学习了如何使用Gradle来构建和运行这个项目。测试代码是软件开发周期中非常重要的一环，能够确保软件的行为能符合预期。这一章我将讲述如何使用Gradle来组织、配置和执行测试代码，学习如何写单元测试、集成测试和功能测试并把他们集成到项目构建中。

Gradle集成了很多Java和Groovy测试框架，在本章的最后你会用JUnit、TestNG和Spock来编写和执行测试，学习控制测试日志的输出、监听测试生命周期事件，以及如何提高测试性能。

自动化测试

如果你想构建可靠的高质量的软件，自动化测试将是你工具箱里面非常关键的一个部分，它帮助你减少手工测试的代价，提高你的开发小组重构已有代码的能力。

自动化测试的类型

并非所有的自动化测试都是相似的，他们通常在作用域、实现方式和执行时间上有所差异，我把他们分成三种类型的测试：单元测试、集成测试和功能测试。

- 单元测试用于测试你代码的最小单元，在基于java的项目中这个单元就是一个方法(method)，在单元测试中你会避免与其他类或者外部的系统打交道。单元测试很容易编写，执行起来非常快速，能够在开发阶段给你代码的正确性提供反馈。
- 集成测试用于测试某一个组件或者子系统。你想确保不同类之间的交互能够按照预期一样，一个典型的情况就是逻辑层需要和数据库打交道。因此相关的子系统、资源文件和服务层必须在测试执行阶段是可访问的。集成测试通常比单元测试运行更慢，更难维护，出现错误时也比较难诊断。
- 功能测试用于测试一个应用的功能，包括和外部系统的交互。功能测试是最难实现也是运行最慢的，因为他们需要模仿用户交互的过程，在web开发的情况下，功能测试应该包括用户点击链接、输入数据或者在浏览窗口提交表单这些情形，因为用户接口可能随着时间改变，功能测试的维护将会很困难。

自动化测试金字塔

你可能想知道到底哪一种测试最适合你的项目，在现实环境中你可能会混合使用这几种测试方法来确保你的代码在不同架构层面都是正确的。你需要写多少测试取决于编写和维护测试的时间消耗。测试越简单就越容易执行，一般来讲你的项目应该包含很多单元测试，少量的集成测试以及更少的功能测试。

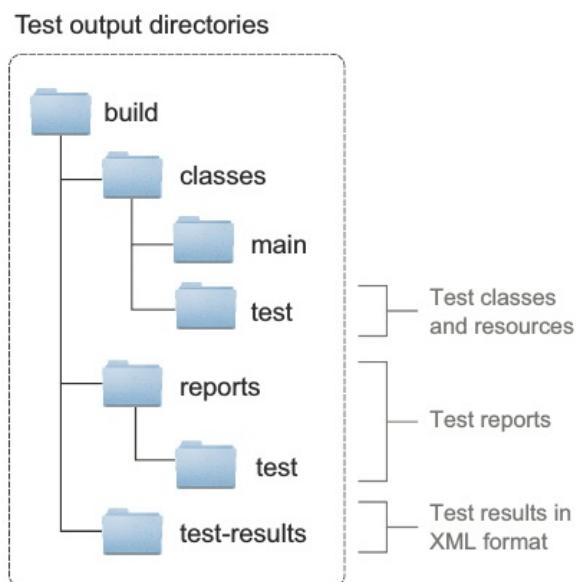
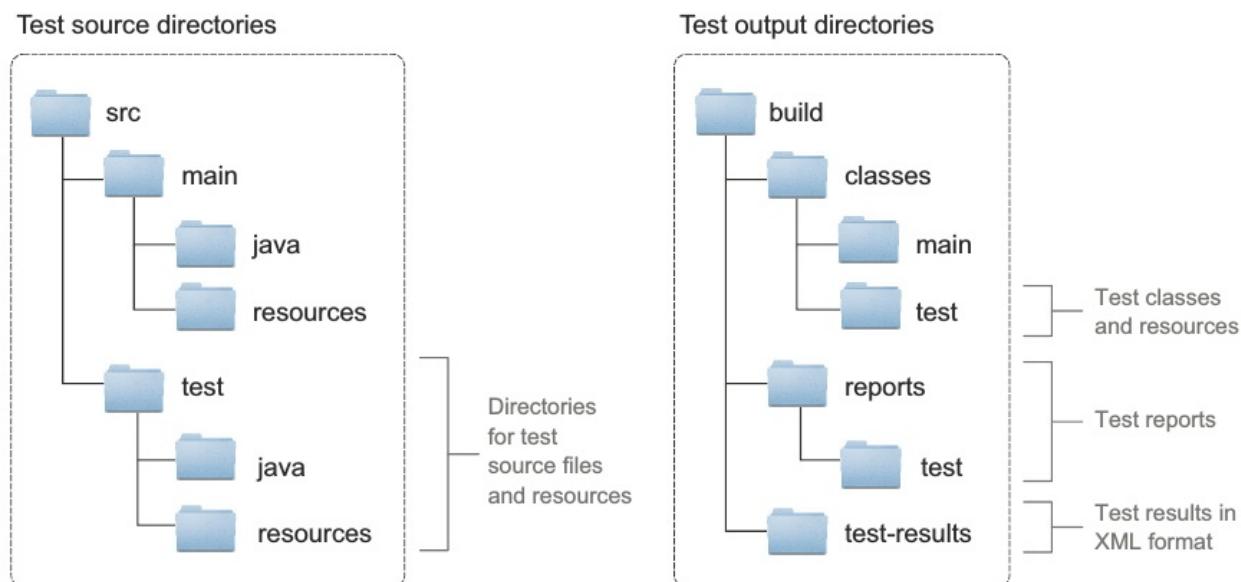
测试Java应用

一些开源的测试框架比如JUnit,TestNG能够帮助你编写可复用的结构化的测试，为了运行这些测试，你要先编译它们，就像编译源代码一样。测试代码的作用仅仅用于测试的情况，你可不想把你的测试代码发布到生产环境中，把源代码和测试代码混在一起可不是个好主意。通常你会把源代码和测试代码分开来，比如Gradle的标准项目布局src/main/java和src/test/java。

项目布局

在前面我们讲到默认的项目布局，源代码是src/main/java，资源文件是在src/main/resources，测试源代码路径也是这样，你把测试代码放在src/test/java，资源文件放在src/test/resources，编译之后测试的class文件在build/classes/test下。

所有的测试框架都会生成至少一个文件用以说明测试执行的结果，最普遍的格式就是XML格式，你可以在build/test-results路径下找到这些文件，XML文件的可读性比较差，许多测试框架都允许把测试结果转换成报告，比如JUnit可以生成HTML格式的报告，Gradle把测试报告放在build/reports/test。下图清晰的显示了项目的布局：



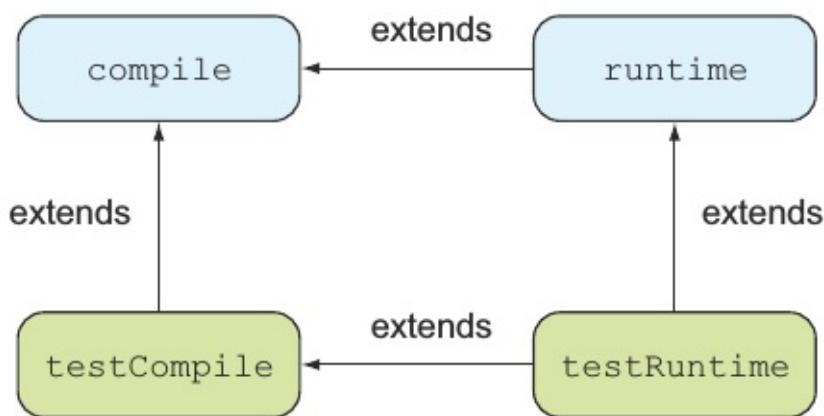
上面讲了这么多测试框架，Gradle怎么知道你想使用哪一个呢？你需要声明对外部库的依赖。

测试配置

Java插件引入了两个配置来声明测试代码的编译期和运行期依赖:testCompile和testRuntime，我们来看一下怎么声明一个对JUnit框架的编译期依赖：

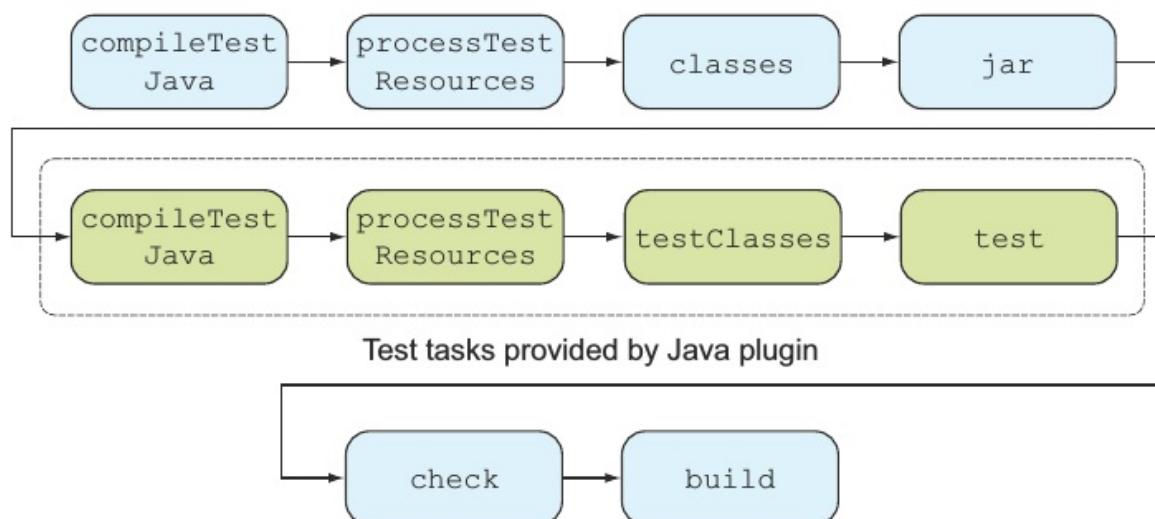
```
dependencies {
    testCompile 'junit:junit:4.11'
}
```

另外一个配置testRuntime用来声明那些编译期用不着但是在运行期需要的依赖，记住用于测试的依赖不会影响你源代码的classpath，换句话说他们不会用在编译或打包过程。然而，对于处理依赖来讲测试配置继承了源代码相关配置，比如testCompile继承了compile配置的依赖，testRuntime继承了runtime和testCompile和他们的父类，他们父类的依赖会自动传递到testCompile或testRuntime中。如下图所示：



测试任务

在之前的任务我们可能注意到任务图一直有四个任务是up-to-date的然后被跳过了，这是因为你没有编写任何测试代码Gradle就不需要编译或执行。下图显示了这四个任务在任务图中的位置：



从图中可以看到测试编译和测试执行阶段是在源代码被编译和打包之后的，如果你想避免执行测试阶段你可以在命令行执行gradle jar或者让你的任务依赖jar任务。

自动测试检查

对于build/classes/test目录下的所有编译的测试类，Gradle怎么知道要执行哪一个呢？答案就是所有匹配下面几条描述的都会被检查：

- 任何继承自junit.framework.TestCase 或groovy.util.GroovyTestCase的类
- 任何被@RunWith注解的子类
- 任何至少包含一个被@Test注解的类

如果没有找到符合条件的，测试就不会执行，接下来我们会使用不同框架来编写单元测试。

单元测试

作为一个Java开发者，你有很多个测试框架可选，这一节我将介绍传统的JUnit和TestNG,如果你没有接触过这些框架，你可以先看看他们的在线文档。

使用JUnit

你将给你之前的ToDo应用的存储类InMemoryToDoRepository.java编写单元测试，为了突出不同框架的相同和不同之处，所有的单元测试都会验证同一个类的功能。接下来你给子项目repository编写测试，放置测试代码的正确位置是在测试的标准布局里，在src/test/java目录下创建一个名叫InMemoryToDoRepositoryTest.java的类，你可以学习测试驱动开发的相关理论，在代码中添加适当的断言语句，下面这段代码用来测试插入功能的正确性。

```
import com.manning.gia.todo.model.ToDoItem;
import org.junit.Before;
import org.junit.Test;
import java.util.List;
import static org.junit.Assert.*;

public class InMemoryToDoRepositoryTest {
    private ToDoRepository inMemoryToDoRepository;
    //用这个注解标识的都会在类的所有测试方法之前执行
    @Before
    public void setUp() {
        inMemoryToDoRepository = new InMemoryToDoRepository();
    }
    //用这个注解的都会作为测试用例
    @Test
    public void insertToDoItem() {
        ToDoItem newToDoItem = new ToDoItem();
        newToDoItem.setName("Write unit tests");
        Long newId = inMemoryToDoRepository.insert(newToDoItem);           //错误的断言会导致测
        assertNull(newId);
        ToDoItem persistedToDoItem = inMemoryToDoRepository.findById(newId);
        assertNotNull(persistedToDoItem);
        assertEquals(newToDoItem, persistedToDoItem);
    }
}
```

接下来你需要在依赖配置中添加JUnit的依赖：

```
project(':repository')repositories {
    mavenCentral()
}
{
}
dependencies {
    compile project(':model')
    testCompile 'junit:junit:4.11'
}
```

之前我们讲过test任务会先编译源代码，生成Jar文件，然后编译测试代码最后执行测试，下面的命令行输出显示了有一个断言出错的情况：

```
$ gradle :repository:test
:model:compileJava
:model:processResources UP-TO-DATE
:model:classes
:model:jar
:repository:compileJava
:repository:processResources UP-TO-DATE
:repository:classes
:repository:compileTestJava
:repository:processTestResources UP-TO-DATE
:repository:testClasses
:repository:test

com.manning.gia.todo.repository.InMemoryToDoRepositoryTest
> testInsertToDoItem FAILED//出错方法的名字
java.lang.AssertionError at InMemoryToDoRepositoryTest.java:24
//测试结果概括
1 test completed, 1 failed
:repository:test FAILED

FAILURE: Build failed with an exception.

* What went wrong:
Execution failed for task ':repository:test'.
> There were failing tests. See the report at:
- file:///Users/ben/dev/gradle-in-action/code/chapter07/junit-test-
- failing/repository/build/reports/tests/index.html
```

从输出可以看出一个断言失败了，这正是你想看到的，显示的信息并没有告诉你为什么测试失败了，指示告诉你第24行的断言失败了，如果你有很多个测试，你需要打开测试报告才能找到出错的原因，你可以在任务使用-i选项打印日志输出：

```
$ gradle :repository:test -i
...
com.manning.gia.todo.repository.InMemoryToDoRepositoryTest
> testInsertToDoItem FAILED
java.lang.AssertionError: expected null, but was:<1>
at org.junit.Assert.fail(Assert.java:88)
at org.junit.Assert.failNotNull(Assert.java:664)
at org.junit.Assert.assertNull(Assert.java:646)
at org.junit.Assert.assertNull(Assert.java:656)
at com.manning.gia.todo.repository.InMemoryToDoRepositoryTest
->.testInsertToDoItem(InMemoryToDoRepositoryTest.java:24)
...
```

在堆栈树我们可以找到出错的原因是newId的值我们假定是null但实际上为1，所以断言出错了，修改之后再运行可以看到所有测试都通过了：

```
$ gradle :repository:test
:model:compileJava
:model:processResources UP-TO-DATE
:model:classes
:model:jar
:repository:compileJava
:repository:processResources UP-TO-DATE
:repository:classes
:repository:compileTestJava
:repository:processTestResources UP-TO-DATE
:repository:testClasses
:repository:test
```

Gradle可以生成更加视觉化的测试报告，你可以在build/reports/test目录下找到HTML文件，打开HTML文件你应该可以看到类似这样的东西：



使用其他测试框架

你可能在你的项目中想使用其他的测试框架比如TestNG和Spock

使用testNG

比如你想用testNG来编写相同的测试类，相似的，你用testNG指定的注解来标识相应的方法，要想你的构建执行testNG测试，你需要做两件测试：

1. 声明对testNG库的依赖
2. 调用Test#useTestNG()方法来声明测试过程使用testNG框架

如下图所示来配置脚本文件：

```
project(':repository') {
    repositories {
        mavenCentral()
    }

    dependencies {
        compile project(':model')
        testCompile 'org.testng:testng:6.8'
    }
    //设置使用testNG来测试
    test.useTestNG()

}
```